Facultade de Informática
UNIVERSIDADE DA CORUÑA

TRABALLO FIN DE GRAO
GRAO EN ENXEÑARÍA INFORMÁTICA
MENCIÓN EN ENXEÑARÍA DO SOFTWARE

Desarrollo del broker AXUDOT para resolución automatizada de solicitudes de autoservicio mediante fuerza de trabajo robótica

Estudante: Iván González Dopico
Dirección: Laura Milagros Castro Souto
Dirección: Jorge Rodríguez Graña

A Coruña, xuño de 2021.

A Carmen y Manuel

Agradecimientos

En primer lugar, agradecer a mis tutores, Laura y Jorge, toda la orientación y ayuda que me han prestado durante la realización de este trabajo.

A Everis, por la confianza y la oportunidad que me han brindado con este proyecto; y a todo el equipo de AXUDOT, por su gran acogida y apoyo constante.

A mi familia y amigos, por animarme en las etapas más difíciles y creer en mi en todo momento. Sin vosotros no lo habría conseguido.

En definitiva, a todas las personas que han estado en mi vida a lo largo de esta etapa y que, de una forma u otra, han colaborado a que esto sea posible.

Resumen

Actualmente, organizaciones de todos los sectores productivos, públicos y privados, están inmersas en programas de transformación digital. En este sentido, la potenciación del uso y capacidades de los canales digitales para la comunicación con los usuarios y clientes es una de las necesidades clave a resolver.

En este proyecto se diseñará y construirá un *broker* de solicitudes de autoservicio, el cual encaminará y orquestará las solicitudes que los usuarios finales realizan desde los sistemas *frontend* de la organización hacia los sistemas *backend* encargados de su resolución; permitiendo, además, reasignar cierta carga de trabajo hacia fuerza humana.

Abstract

Nowadays, organizations from all productive sectors, both public and private, are immersed in digital transformation programs. In this context, the enhancement of the use and capabilities of digital channels for communication with users and customers is one of the key needs to be addressed.

In this project, a self-service request broker will be designed and built, which will route and orchestrate the requests that end users make from the organization's frontend systems to the backend systems in charge of their resolution. This will allow, in addition, for the reallocation of a certain load of work towards human workforce.

Palabras clave:

- Automatismo
- *Broker* de mensajería
- Solicitud
- Orquestación de flujo
- Sistema resolutor
- Scrum
- Cola
- Mensaje

Keywords:

- Automatism
- Message broker
- Request
- Flow orchestration
- Resolver system
- Scrum
- Queue
- Message

Índice general

Índice de figuras

Índice de tablas

Capítulo 1

Introducción

En este capítulo se expondrá la motivación, contexto y objetivos del proyecto, junto a la estructura que seguirá el presente documento.

1.1 Motivación

Vivimos en una sociedad cada vez más dominada por las nuevas tecnologías. Su aparición ha cambiado nuestra forma de afrontar las necesidades del día a día, de relacionarnos y, por supuesto, de trabajar.

Como consecuencia de ello, numerosas organizaciones de todos los ámbitos y sectores se encuentran actualmente involucradas en el procedimiento de incorporar estas tecnologías dentro de su método de trabajo. Mediante este proceso, conocido como transformación digital, la organización busca aprovechar al máximo los beneficios que proporcionan las nuevas herramientas digitales y los nuevos modelos de negocio asociados a las mismas.

Una de las tareas más interesantes a abordar, y en la que se centrará este proyecto, es la automatización de solicitudes de autoservicio y procesos administrativos. Como se puede apreciar en la Figura 1.1, una gran parte de la jornada laboral de los empleados es dedicada a tareas administrativas que requieren procesamiento manual. Esto provoca una pérdida considerable de tiempo tanto para la empresa como para el cliente o empleado que realiza la solicitud, lo que se ve traducido en una pérdida de productividad y, finalmente, de rentabilidad.

Este proyecto busca mitigar este problema, ofreciendo una solución liberada de la intervención humana, mediante el desarrollo de una plataforma que encaminará las solicitudes a sus respectivos sistemas resolutores; reduciendo así considerablemente los tiempos de espera y aumentando notablemente la capacidad de respuesta y la productividad de la organización.

1.2 Contexto

Este trabajo nace como una necesidad durante el desarrollo del proyecto AXUDOT, que será nuestro caso de estudio. AXUDOT es una plataforma (Figura 1.2) diseñada para facilitar un entorno digital en el que poder realizar diversas solicitudes destinadas a distintos organismos, revisar el estado de las mismas o aceptar/rechazar las que están bajo nuestra aprobación. Todo esto sin necesidad de intermediación humana y automatizando (siempre que sea posible) todo el proceso. Y precisamente en esto último es en lo que nos centraremos.

1.3 Objetivos

La finalidad de este proyecto es desarrollar un *broker* de autoservicio mediante un componente software, el cual recibirá las solicitudes generadas desde un portal web y las procesará con los sistemas resolutores. Dicho componente también incluirá la lógica correspondiente a las aprobaciones; así como la necesaria para derivar las solicitudes no automatizables al soporte humano.

En concreto, el componente dará soporte a tres sistemas resolutores distintos (pág. 24), los cuales se integrarán siguiendo una serie de requisitos; y la comunicación con estos se realizará mediante el uso de colas (pág. 24), en las cuales se depositarán los mensajes a la espera de que sean consumidos.

Los principales objetivos a cumplir durante el desarrollo de este proyecto son:

- **Recepción de solicitudes:** Recibir y recoger las solicitudes generadas desde *frontend* a través del portal AXUDOT.
- **Orquestación de flujos de mensajería:** Diferenciar los mensajes y orientarlos hacia el flujo de automatización adecuado en función de su código y estrategia asociados.
- **Lógica de aprobaciones y de derivación hacia soporte humano:** Establecer los criterios para que una solicitud sea automatizable, así como los casos en los que sea necesario delegar en soporte humano.
- **Conectores con sistemas resolutores:** Enviar las solicitudes a sus correspondientes sistemas resolutores y delegar su resolución en estos.
- **Monitorización y notificación del estado de la cola de automatismos:** Supervisar y revisar periódicamente la cola de automatismos y sus mensajes para poder tener un control y seguimiento de los mismos, pudiendo comprobar así si hay algún mensaje que se haya quedado atascado y/o sin ser procesado.

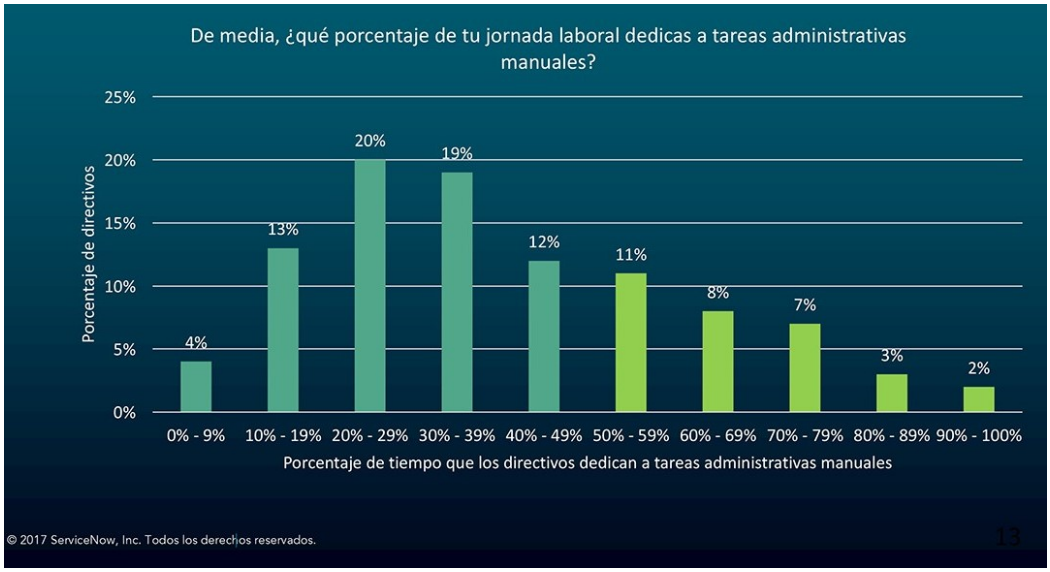


Figura 1.1: Porcentaje diario dedicado a tareas administrativas manuales

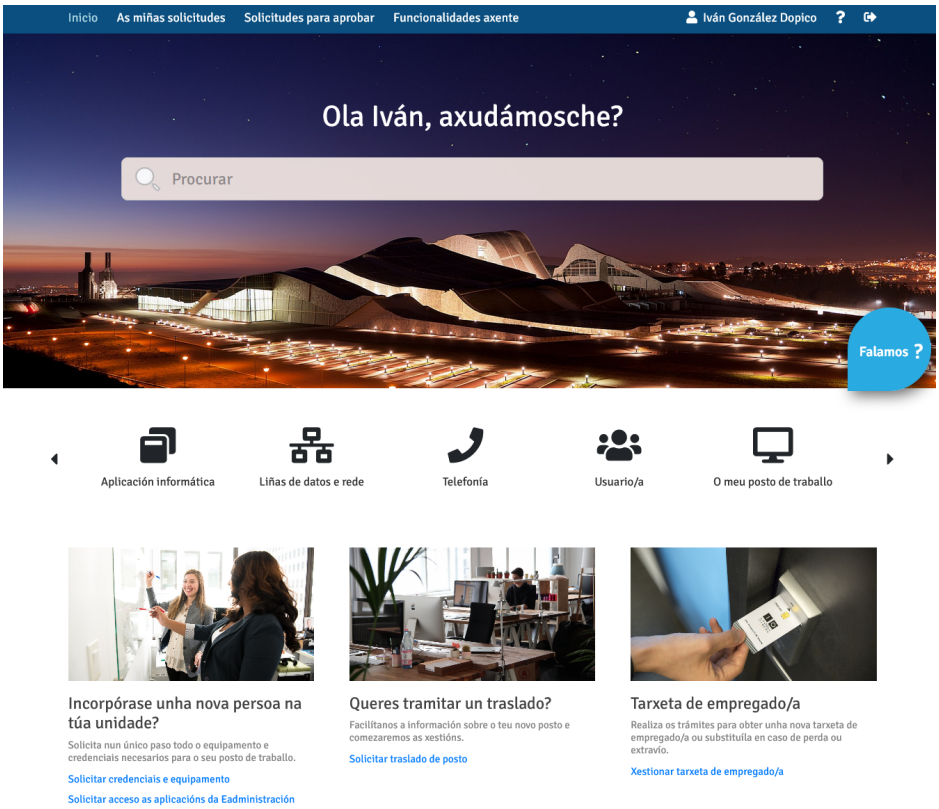


Figura 1.2: Página de inicio de la plataforma AXUDOT

1.4 Estructura de la memoria

- **Introducción:** Expone el contexto en el que se desarrolla el proyecto, junto con su motivación y los objetivos del mismo. Además, detalla la estructura que seguirá la memoria.
- **Fundamentos tecnológicos:** Enumera y explica brevemente las tecnologías usadas durante el desarrollo.
- **Metodología:** Explica la metodología escogida para la realización del proyecto y su adaptación al mismo, junto a la planificación por parte del alumno.
- **Conocimientos previos:** Detalla los conceptos necesarios para la correcta comprensión de este documento,
- **Requisitos y arquitectura:** Presenta al análisis de los requisitos del proyecto, junto a las decisiones y justificaciones de diseño y el modelo de datos, así como los casos de uso más destacados.
- **Desarrollo:** Recoge todo el proceso seguido para el desarrollo de los módulos principales del sistema, detallando su diseño e implementación.
- **Pruebas:** Describe las pruebas llevadas a cabo para comprobar que el sistema tiene el comportamiento deseado.
- **Conclusiones:** Presenta el análisis y la valoración final del proyecto, atendiendo al nivel de cumplimiento de los requisitos y la mejora obtenida en las medidas; así como los conocimientos aplicados y el trabajo futuro.

Capítulo 2

Fundamentos tecnológicos

En este apartado se exponen las distintas herramientas y tecnologías empleadas para llevar a cabo este proyecto. La elección de las mismas se debe a que eran las que ya se estaban empleando en el proyecto AXUDOT al comienzo del presente trabajo.

2.1 Lenguajes de programación

2.1.1 Java

Java [1] es un lenguaje orientado a objetos, multiplataforma y basado en clases. Dispone de una gran compatibilidad con numerosas herramientas y *frameworks*, entre ellos Spring, el cual se empleará en el desarrollo del proyecto.

2.1.2 SQL

SQL es un lenguaje estructurado de consulta desarrollado por IBM, empleado para realizar peticiones de consulta y administración sobre bases de datos[2] relacionales.

2.2 Frameworks

2.2.1 Spring

Spring[3] es un framework de código abierto para el desarrollo de aplicaciones web en Java. Permite desarrollar aplicaciones con mayor rapidez, facilidad y seguridad. Su enfoque modular permite la utilización de uno de estos módulos sin necesidad de emplear los demás. En este proyecto se hace uso de Spring Data JPA, que facilita la implementación de repositorios basados en Java Persistence API (JPA).

2.3 Herramientas de desarrollo

2.3.1 Apache Maven

A lo largo del desarrollo de software, es necesario hacer numerosas compilaciones, ejecuciones y despliegues (entre otras cosas), lo cual termina suponiendo un gasto de tiempo y esfuerzo considerables.

Apache Maven [4] proporciona una solución a esto. Basando su funcionamiento en un archivo XML llamado Project Object Model (POM), en el que se gestionan las distintas fases de construcción y dependencias, se encarga de automatizar y simplificar todo este proceso.

2.3.2 Apache ActiveMQ

Apache ActiveMQ [5] es un proyecto de software libre que proporciona un sistema de mensajería asíncrono, fiable y de alto rendimiento, un mecanismo imprescindible para este proyecto. Más adelante se analizarán en sus características y posibilidades con mayor detalle (página 18).

2.3.3 IntelliJ IDEA

IntelliJ IDEA [6] es un entorno de desarrollo integrado multilenguaje, diseñado para maximizar la productividad. Está desarrollado por JetBrains y ofrece dos versiones: una para la comunidad (la que se empleará) y otra comercial.

Su característica más destacable es la gran asistencia que ofrece al usuario, aportando sugerencias y atajos de finalización de código de manera instantánea e inteligente, y realizando un análisis de código al momento.

Tiene soporte para múltiples lenguajes, destacando la gran integración que facilita con proyectos basados en Maven; y también permite la conexión con sistemas de versionado, lo que lo convierte en un entorno perfecto para este desarrollo.

2.3.4 SQL Developer

Para el desarrollo y mantenimiento de la base de datos, se ha decidido emplear SQL Developer [7].

Es un entorno de desarrollo diseñado para el trabajo con bases de datos de Oracle, que permite realizar un desarrollo tanto de forma local como en la nube.

Proporciona hojas de trabajo para ejecutar scripts, consolas para gestionar la base de datos y una interfaz que nos permite navegar entre las distintas tablas con un solo clic.

2.3.5 Apache Subversion

Apache Subversion [8] es un sistema de control de versiones de código libre que basa su funcionamiento en el concepto de revisión.

Con cada nueva revisión, Subversion solo guarda las modificaciones entre dicha revisión y la anterior, optimizando así el espacio en disco, y permitiendo hacer *rollbacks* de forma sencilla.

Las funciones permitidas al usuario son muy similares a las del sistema de archivos de cualquier sistema operativo, y permite el acceso a través de la red, por lo que es posible manejar los archivos desde distintos equipos de forma coordinada.

2.3.6 Docker

Docker [9] es un proyecto de software libre diseñado para la automatización del despliegue dentro de un entorno aislado conocido como contenedor. Este aislamiento ofrece la posibilidad de desplegar diversos contenedores en un solo *host*.

Para llevar esto a cabo, se hace uso de Docker Compose [10], una herramienta que, a partir de un archivo YAML que contiene la configuración necesaria, es capaz de definir y ejecutar aplicaciones con diversos contenedores.

2.3.7 Jenkins

La integración continua es una práctica de ingeniería software la cual consiste en realizar la compilación y ejecución de pruebas de un proyecto de forma periódica y automatizada, con el fin de detectar fallos lo antes posible.

Para llevar a cabo esta tarea se ha optado por el uso de Jenkins [11], un servidor de código abierto que permite realizar lo expuesto anteriormente, y también facilita la tarea de despliegue. Además, permite llevar a cabo todo este proceso con una integración total con nuestro control de versiones, lo que agiliza mucho el desarrollo.

2.3.8 Redmine

Otra parte fundamental en el desarrollo de un proyecto software es tener una buena gestión del mismo, y para conseguir esto se ha mantenido un registro de las tareas realizadas empleando Redmine [12].

Esta herramienta permite llevar un seguimiento y un control del estado del proyecto, así como asignar tareas a sus desarrolladores, dividirlas por sprints, obtener gráficas de la evolución de las mismas, etc.

Capítulo 3

Metodología

En este capítulo se explica brevemente la metodología empleada y su aplicación y adaptación en el proyecto. Además, se expone la planificación del trabajo por parte del alumno.

3.1 Scrum

Scrum[13, 14] es un marco de trabajo nacido en los años 80 e ideado para seguir un desarrollo ágil en proyectos software. Cabe aclarar que Scrum no es un único proceso o técnica, sino que define un conjunto de estas mismas, adaptado a las necesidades y exigencias de cada proyecto. La idea principal sobre la que pivota Scrum es la auto-organización del equipo, objetivo para el cual se proporcionan distintos roles dentro de los mismos; y se basa en la teoría del empirismo, en la que el conocimiento parte de la experiencia previa y de lo ya conocido. Propone un desarrollo incremental de funcionalidades, en la que cada fase intermedia proporciona un producto entregable y funcional.

3.2 Aplicación al caso de estudio

Durante el desarrollo del proyecto, se ha seguido una aproximación de la metodología Scrum adaptada a las necesidades específicas del mismo, con las siguientes características:

- Daily Scrum: Es una breve reunión diaria (unos 15 minutos) en la que el Equipo de Desarrollo pone en común el trabajo realizado en las últimas 24 horas. Cada miembro del equipo dará respuesta a las siguientes preguntas:

    1. ¿Qué hice desde el último Scrum?
    2. ¿Qué haré hasta el próximo Scrum?
    3. ¿Detecto algún impedimento en el desarrollo del trabajo?

Se realizan siempre a la misma hora siendo previamente agendadas, y en ellas se abordan los apartados típicos de estas reuniones. Están presentes el tutor del proyecto, cumpliendo el rol de Scrum Master, y el Equipo de Desarrollo, del cual forma parte el alumno, quién participará como un miembro más.

- Product Backlog: Es una lista, ordenada según la prioridad, de todas las funcionalidades técnicas y de negocio que podría ser necesario desarrollar para el producto. Es un artefacto dinámico, es decir, cambia junto al producto y su entorno. Por ello, nunca se puede considerar completo, y su vida va ligada a la del producto: mientras éste exista, su Product Backlog también lo hará. A medida que van surgiendo nuevos requisitos o modificaciones en el proyecto, ya sea a petición del cliente o como consecuencia del desarrollo, se crea una nueva tarea y se añade al Product Backlog. Para ello se emplea la herramienta Redmine (ver Figura 3.1), asignándose las mismas al desarrollador encargado de llevarla a cabo. Éste irá actualizándola indicando el estado, porcentaje completado o posibles tareas relacionadas, modificando la descripción y añadiendo notas cuando sea necesario, etc.

- Sprint Backlog: Está formado por los elementos del Product Backlog seleccionados para el Sprint. Estas tareas están acompañadas de sus correspondientes estimaciones y planes de trabajo. En este caso el encargado de su gestión es el Equipo de Desarrollo, que podrá añadir o modificar tareas a lo largo del Sprint según sea necesario. Gestionado también con Redmine, se etiquetan las tareas del Product Backlog que se van a abordar en el Sprint actual, diferenciándolas así de las demás y etiquetándolas con la versión correspondiente al final de cada Sprint.

Figura 3.1: Ejemplo de tarea gestionada con Redmine

3.3 Planificación

Como ya se comentó con anterioridad, este proyecto se encuentra enmarcado en el contexto de uno mayor, AXUDOT. Esto ha afectado a la planificación, ya que, debido a la necesidad de seguir desarrollando otros apartados de forma paralela, los Sprints no han podido ser dedicados de forma exclusiva a las tareas de automatización que afectan a este trabajo. De la misma forma, la planificación de estos dependía en gran medida de dichos desarrollos y su necesidad de pase a entornos de producción. Por lo tanto, en lugar de una clásica división en Sprints y explicación de cada uno de los mismos, se ha decidido exponer la planificación desde la perspectiva del alumno y su organización y dedicación en cada una de las tareas abordadas.

El primer mes de desarrollo se dedicó a la configuración de todas las herramientas y entornos necesarios para el desarrollo del proyecto. Además, durante este período el alumno recibió formación de las bases del sistema global y se familiarizó con el mismo.

Las siguientes cuatro semanas fueron dedicadas a la formación en la herramienta ActiveMQ y a la valoración de las distintas opciones de diseño para el sistema de colas, junto con su decisión final. En este momento el alumno ya pasa a integrarse completamente en la dinámica de trabajo del equipo de desarrollo.

A continuación, se abordaron las tareas correspondientes a las decisiones de flujo de mensajería y lógicas de aprobaciones. Al estar relacionadas con indicaciones de los clientes y no requerir un tiempo muy grande de dedicación, se decidió implementar en paralelo el servicio de activación/desactivación de los automatismos desarrollados paralelamente. Este trabajo en conjunto requirió cerca de 5 semanas de trabajo.

En este punto del desarrollo, surgió la necesidad de realizar un control sobre los *tickets* pendientes de automatizar que se hayan podido quedar atascados en la cola de automatismos propia de la herramienta de *ticketing*. Se decide que el alumno se encargue de su desarrollo, por lo que dedica una primera semana a documentarse sobre las distintas opciones de diseño y sobre cómo se podrían realizar notificaciones diarias del estado de dicha cola. Una vez presentadas las distintas opciones al resto del equipo, lo cual ocupa otra semana a mayores, se lleva a cabo la implementación de la solución escogida. Debido a necesidades de coordinación con distintas secciones de la empresa para la habilitación de herramientas y configuraciones necesarias para el funcionamiento del servicio, su desarrollo se extiende otras tres semanas.

Las últimas tres semanas del proyecto se dedicaron a la integración de un nuevo sistema resolutor y a la elaboración y extensión de las distintas pruebas de unidad y de integración de los componentes implementados.

Cabe destacar que durante todo el desarrollo se trabajó paralelamente en la memoria, sobre todo en el diseño y planificación de la misma y sus potenciales contenidos. Fue en las seis últimas semanas del proyecto cuando se hizo hincapié en la redacción de la memoria,

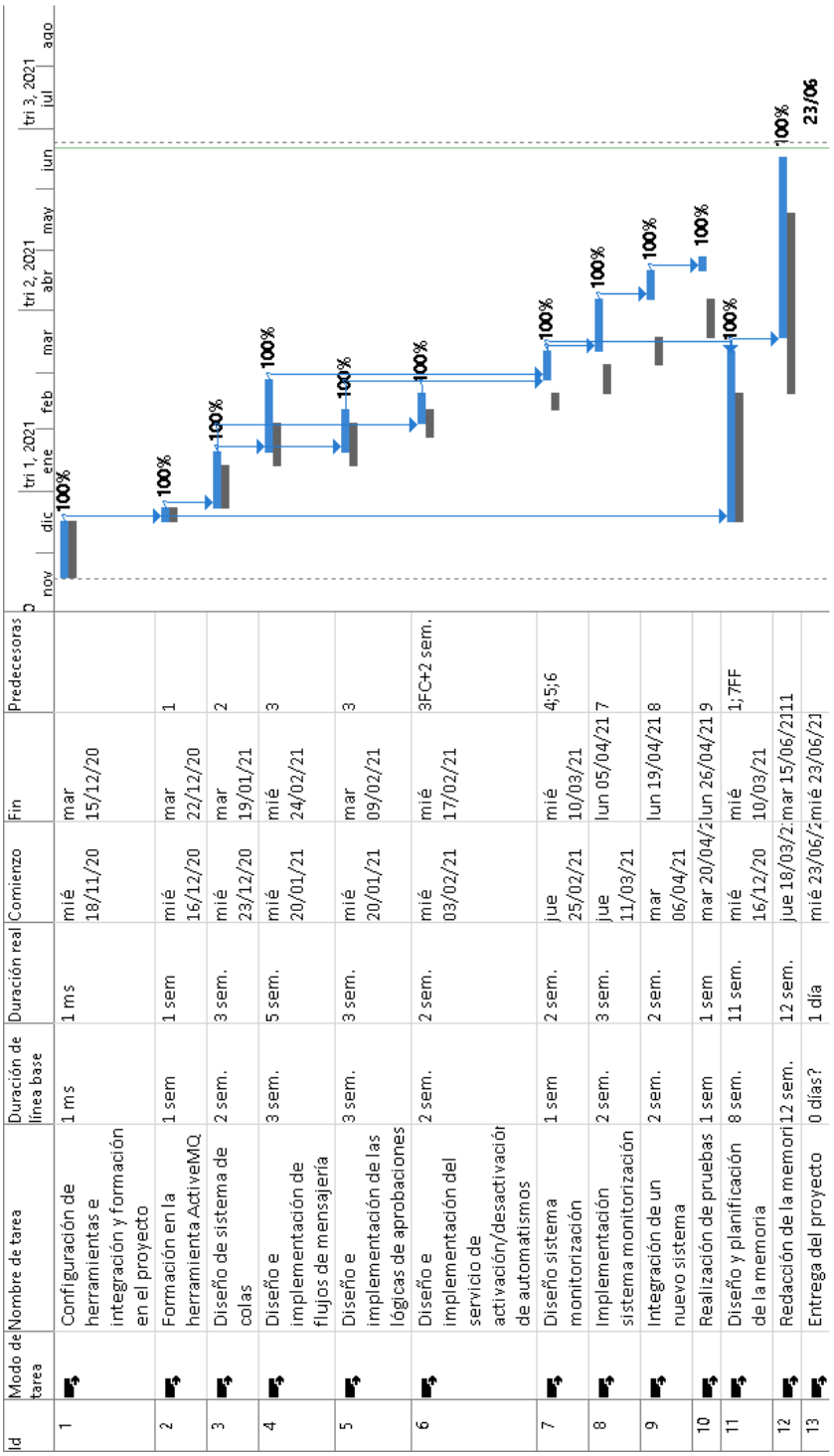
extendiéndose esta tarea seis semanas extra tras la finalización del desarrollo.

En total, el tiempo dedicado a la realización de este trabajo ha sido de 27 semanas (noviembre 2020 - junio 2021). Esto, teniendo en cuenta que los primeros cuatro meses se dedicó una jornada de 4 horas de trabajo, y los tres restantes una jornada de 6 horas, equivale a un total de 650 horas de trabajo. Si se emplea como referencia el salario medio para un Ingeniero Informático que acaba de terminar sus estudios reflejado en un estudio de la Universidad Europea[1], con valor de $10'42$€/h[2], se obtiene un coste total de 6.773€.

Esta planificación puede observarse en su conjunto en el diagrama de Gantt correspondiente (Figura 3.2), en el cual se pueden apreciar las relaciones entre las tareas, junto con sus estimaciones iniciales y correspondientes retrasos. Se indica como hito final la fecha de entrega del trabajo.

[1] https://universidadeuropea.com/blog/cuanto-gana-un-ingeniero-informatico

[2] Para calcular este valor por hora, se tiene en cuenta el máximo de 1.800 horas de trabajo anuales establecido en el *XVI Convenio colectivo estatal de empresas de consultoría y estudios de mercado y de la opinión pública*[15]. Esto, junto al mínimo de 23 días de vacaciones anuales establecidos en el mismo convenio, nos permite escoger una jornada de 8 horas semanales (160 horas mensuales).



Id	Modo de tarea	Nombre de tarea	Duración de línea base	Duración real	Comienzo	Fin	Predecesoras
1		Configuración de herramientas e integración y formación en el proyecto	1 ms	1 ms	mié 18/11/20	mar 15/12/20	
2		Formación en la herramienta ActiveMQ	1 sem	1 sem	mié 16/12/20	mar 22/12/20	1
3		Diseño de sistema de colas	2 sem.	3 sem.	mié 23/12/20	mar 19/01/21	2
4		Diseño e implementación de flujos de mensajería	3 sem.	5 sem.	mié 20/01/21	mié 24/02/21	3
5		Diseño e implementación de las lógicas de aprobaciones	3 sem.	3 sem.	mié 20/01/21	mar 09/02/21	3
6		Diseño e implementación del servicio de activación/desactivación de automatismos	2 sem.	2 sem.	mié 03/02/21	mié 17/02/21	3FC+2 sem.
7		Diseño sistema monitorización	1 sem	2 sem.	jue 25/02/21	mié 10/03/21	4;5;6
8		Implementación sistema monitorización	2 sem.	3 sem.	jue 11/03/21	lun 05/04/21	7
9		Integración de un nuevo sistema	2 sem.	2 sem.	mar 06/04/21	lun 19/04/21	8
10		Realización de pruebas	1 sem	1 sem	mar 20/04/2	lun 26/04/21	9
11		Diseño y planificación de la memoria	8 sem.	11 sem.	mié 16/12/20	mié 10/03/21	1;7FF
12		Redacción de la memori	12 sem.	12 sem.	jue 18/03/2	mar 15/06/21	11
13		Entrega del proyecto	0 días?	1 día	mié 23/06/2	mié 23/06/21	

Figura 3.2: Diagrama de Gantt de seguimiento del proyecto

Capítulo 4

Conocimientos previos

En este capítulo se explicarán y analizarán detalladamente los conceptos necesarios para la correcta comprensión del trabajo, además de las tecnologías directamente relacionadas con los mismos.

4.1 Arquitectura por capas

La arquitectura por capas es uno de los modelos de desarrollo software más empleados en la actualidad. Su idea principal es que una capa inferior proporcione servicio a otra superior, organizadas de manera que cada una de las capas descanse sobre su inferior. De esta manera, una capa solo es consciente de la existencia de la contigua, e ignora la presencia de las demás. La aplicación de este modelo proporciona una serie de interesantes ventajas:

- Permite independizar el software de cada capa. A la capa superior no le importa la manera en la que la inferior implementa el servicio que le proporciona. Esto implica que:
  - Cualquier cambio en la forma de implementar una de las capas no afecta a la otra.
  - Los desarrolladores no necesitan conocer las tecnologías usadas en las otras capas.
  - Aumenta la facilidad de mantenimiento del software. Los cambios en la implementación de un servicio se aíslan dentro de la capa a la que pertenece, sin afectar al resto.
- Pueden desarrollarse distintas capas en paralelo.
- Facilita la escalabilidad y la tolerancia a fallos. En caso de que un servicio falle, puede reemplazarse su capa por una nueva sin necesidad de modificar las demás. De la misma forma, pueden aumentarse los recursos de una capa en concreto.

- Reusabilidad: una misma capa puede ser empleada por más de un servicio de mayor nivel.

4.1.1 Arquitectura típica

A continuación se describen las capas principales de un patrón de arquitectura por capas:

- Capa de Acceso a Datos o *Persistence*: Como su nombre indica, se encarga de la comunicación con los sistemas encargados del almacenamiento de los datos (generalmente bases de datos). Debe asegurarse de abstraer por completo el dialecto empleado en esta comunicación.

- Capa Lógica de Negocio o *Service*: Implementa la lógica de negocio de la aplicación, empleando información proporcionada por el usuario y datos almacenados. Para ello, hace uso de la capa de Acceso a Datos. Es empleada por la capa Interfaz.

- Capa Interfaz o *Presentation*: Encargada de la interacción entre el usuario y el sistema. Se distinguen dos casos:

  - Usuario humano: Interfaz gráfica que permite a los usuarios acceder a las funcionalidades implementadas por la capa modelo.
  - Otras aplicaciones como usuarias: Interfaz programática orientada a que otras aplicaciones utilicen las funcionalidades de la capa modelo. Generalmente se exponen como servicios web.

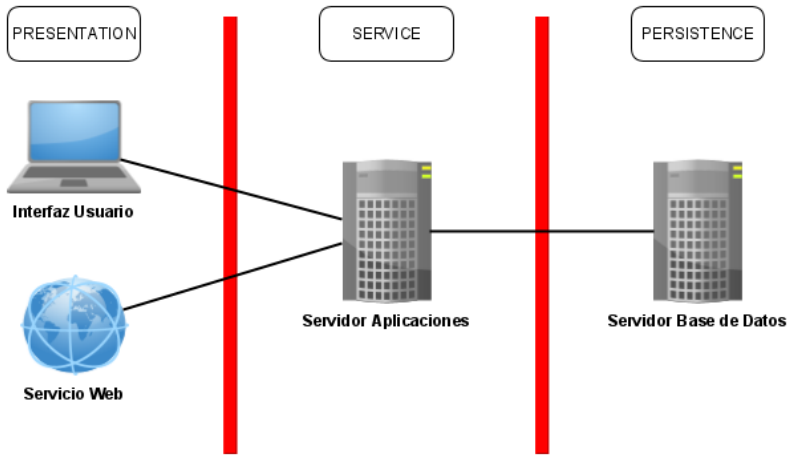


Figura 4.1: Ejemplo de arquitectura en capas

4.2 *Broker* de mensajería

Es un mecanismo de integración de componentes basado en el intercambio de mensajes. Actúa como Message-oriented Middleware (MoM) [1] en la comunicación entre distintos sistemas, permitiendo la transformación de mensajes y la gestión del enrutamiento y distribución de los mismos.

Actúa como intermediario entre los productores de los mensajes (quienes los crean) y los consumidores (quienes los leen y procesan), traduciéndolos al lenguaje formal establecido por cada uno de los mismos. De esta forma, los sistemas comunicantes no necesitan conocerse el uno al otro, ni siquiera estar escritos en el mismo lenguaje o emplear las mismas técnicas, lo que proporciona un gran desacoplamiento.

La mayoría de los proveedores de estos *brokers* proporcionan un almacenamiento persistente de los mensajes. Esto, además de proteger a los mensajes de una posible pérdida, permite una comunicación asíncrona. Los emisores pueden enviar el mensaje y seguir con su trabajo sin preocuparse de que el receptor lo consuma, ya que es el proveedor el que asegura la conservación y enrutamiento del mensaje. Los receptores, una vez listos para consumirlo, se lo solicitan al proveedor y es este el que se lo traslada. Con este tipo de comunicación, se disminuye notablemente el número de operaciones de E/S, incrementando así el rendimiento de nuestro sistema.

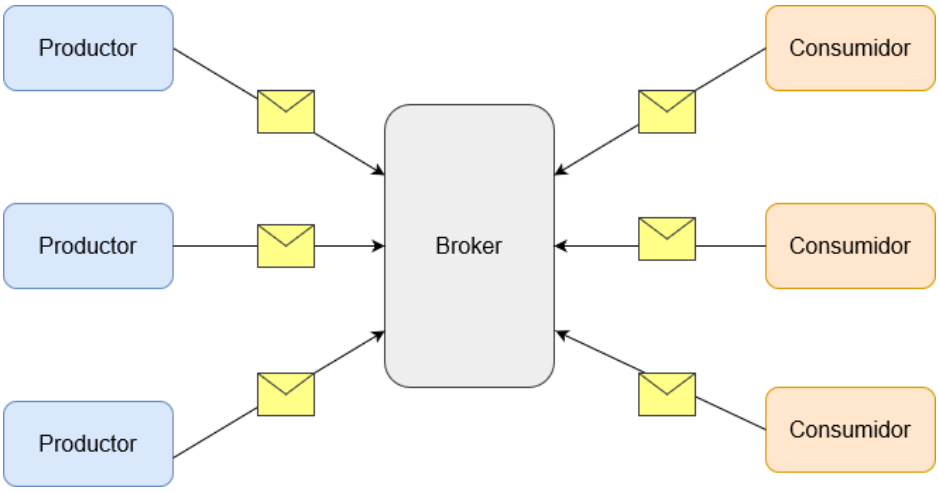


Figura 4.2: Modelo básico de un *broker* de mensajería

Otra de sus principales características es la capacidad de transformar los mensajes. Al emplear este sistema, no es imprescindible que el formato del mensaje que envía el productor sea idéntico al que recibe el consumidor. Cada sistema emplea su propio formato nativo y es

[1] Infraestructura basada en un software o hardware que soporta el envío y recibo de mensajes entre sistemas.

el *broker* el encargado de traducirlo, pudiendo incluso transformar el mismo mensaje a más de un formato distinto, y comunicar así varios sistemas de forma muy sencilla.

4.2.1 Mensajería de colas (Point-to-point)

Tipo de mensajería asíncrona en el cual un cliente (productor) envía el mensaje a una cola, de la que posteriormente será leído por un sistema resolutor (consumidor). Una vez este lo ha recogido, se lo hace saber al servidor y el mensaje desaparece de la cola. En caso de una caída del sistema, los mensajes de los que no se tiene constancia que hayan sido leídos, volverán a estar disponibles para reenviarlos, garantizando así una alta fiabilidad. Permite la existencia de múltiples consumidores en la misma cola, pero cada mensaje solo podrá ser consumido por uno de ellos, lo que garantiza que un mensaje no será leído dos o más veces.

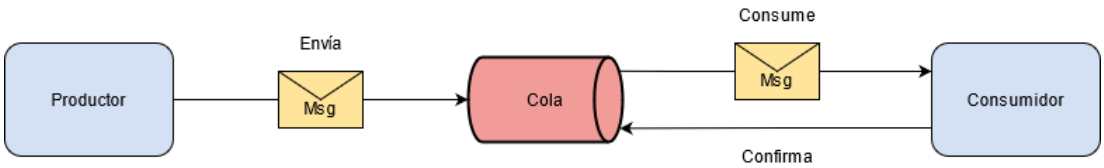


Figura 4.3: Modelo de mensajería Point-to-point

4.2.2 Mensajería Publicador/Suscriptor (Pub/Sub)

Modelo de mensajería asíncrona en el que los mensajes pueden ser enviados por múltiples sistemas (publicadores) hacia el servidor, quedando alojados en una entidad conocida como *topic*. Los consumidores podrán establecer múltiples suscripciones a estas entidades, y cada una de estas suscripciones recibirá una copia del mensaje. Es decir, cada mensaje podrá ser consumido por más de un sistema resolutor, a diferencia de lo que ocurre con el Point-to-point.

Las suscripciones pueden ser duraderas, lo que implica el almacenamiento de una copia de cada mensaje hasta que se consuma, incluso en caso de fallo o reinicio del servidor; o no duraderas, en cuyo caso la suscripción durará tanto como la conexión que la creó, como máximo.

4.3 ActiveMQ Artemis

Es un *broker* de mensajería de código abierto y un claro ejemplo de MoM. Escrito totalmente en Java, implementa por completo la interfaz de Java Message System (JMS)[2], y funciona sobre cualquier plataforma con un entorno Java 8+ (aunque también puede usarse sobre más lenguajes). Será el empleado en este proyecto.

[2] Especificación de Java que define una plataforma de intercambio de mensajes.

Proporciona mensajería persistente a un gran rendimiento, asegura una alta disponibilidad gracias a su *failover*[3] automático para los clientes, y todas las funciones que caben esperar de un sistema de mensajería. Además, permite crear y montar imágenes en Docker de manera sencilla, proporcionando una serie de scripts para ponerlo en marcha.

Actualmente facilita dos implementaciones de su Application Programming Interface (API) para el lado del cliente:

- Core client API: una sencilla e intuitiva API Java alineada con el *core* interno de Artemis. Ofrece un completo conjunto de funcionalidades sin algunas de las complejidades de JMS.
- JMS client API: API estándar de JMS en el lado del cliente. Cuando se emplea esta implementación, todas las interacciones JMS son traducidas al API *Core* de Artemis antes de ser transferidas por medio del protocolo.

4.4 JavaMail

JavaMail es un API Java que proporciona los objetos e interfaces necesarios para establecer un sistema de envío y recepción de correo electrónico a través de los protocolos SMTP, IMAP, MIME y POP3. En nuestro caso concreto, la comunicación se realizará con SMTP.

A continuación, se detallan las clases e interfaces que se emplearán en el proyecto:

- **javax.mail.Session**: Esta clase permite definir una sesión mediante objetos `java.util.Properties`, indicando el host, puerto o protocolo, entre otras propiedades.
- **javax.mail.internet.MimeMessage**: Será la clase empleada para elaborar los mensajes con sus correspondientes atributos. Presenta métodos para establecer la dirección de origen, destinatarios, asunto, contenido, fecha de envío...
- **javax.mail.Transport**: Esta clase es la encargada de realizar el envío del mensaje. Dispone del método `send`, el cual recibe como argumento el mensaje previamente formado, y lo manda al grupo de destinatarios especificado en el mensaje.

[3] Si se produce un fallo de hardware en alguna de las máquinas del clúster, el software de alta disponibilidad es capaz de arrancar automáticamente los servicios en cualquiera de las otras máquinas del clúster.

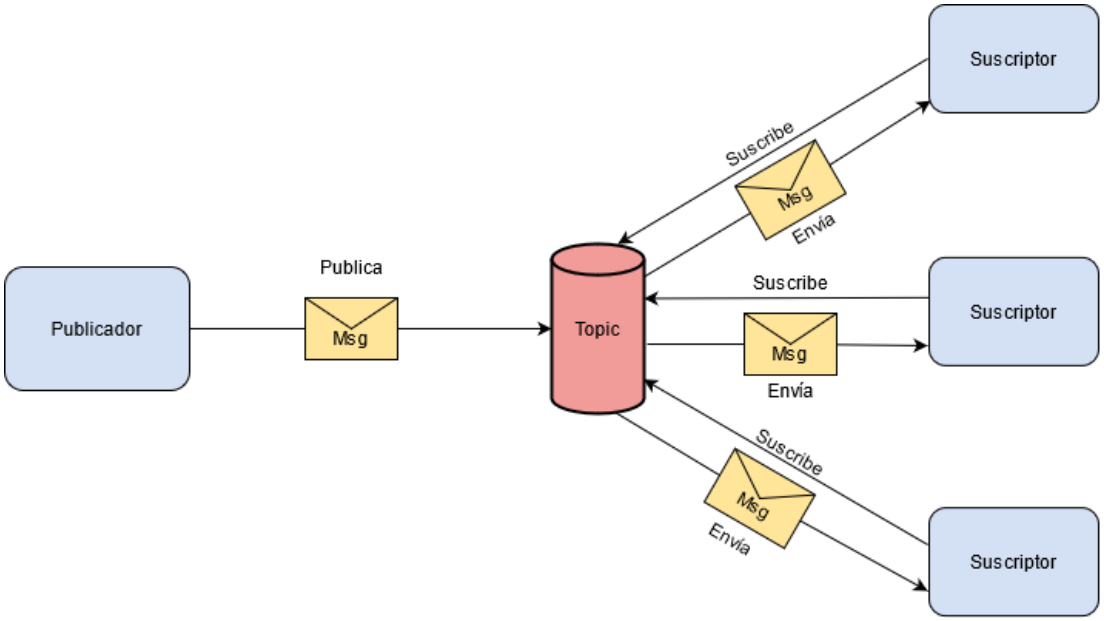


Figura 4.4: Modelo de mensajería Pub/Sub

Capítulo 5

Requisitos y arquitectura

En este capítulo se enumeran los requisitos del proyecto, y se exponen y justifican las decisiones referentes al diseño del sistema y sus componentes. A continuación, se tratan los casos de uso más destacados y, finalmente, se detalla el modelo de datos y las tablas que lo forman.

5.1 Análisis de requisitos

Un buen análisis de requisitos es fundamental para el correcto desarrollo de un proyecto software. Nos permitirá tener una visión clara de las demandas del cliente y los objetivos a cumplir.

En esta sección se especificarán los requisitos identificados en este proyecto, tanto funcionales como no funcionales.

5.1.1 Requisitos funcionales

Como su nombre indica, estos requisitos representan las funcionalidades que debe proporcionar el sistema. Los correspondientes a este proyecto se pueden observar en la Tabla 5.1.

5.1.2 Requisitos no funcionales

Estos requisitos, a diferencia de los funcionales, no definen funciones a realizar ni servicios que proporcionar, sino que reflejan propiedades y características de funcionamiento. Por lo tanto, serán muy relevantes a la hora de tomar decisiones futuras de diseño e implementación.

Los requisitos no funcionales que se han identificado para este proyecto se pueden observar en la Tabla 5.2.

Código	Nombre	Descripción
RF-1	Lectura de *tickets*	Recogida y lectura de los *tickets* generados desde *frontend*
RF-2	Filtro de automatización	Determinar si un *ticket* es apto para automatizar o no.
RF-3	Derivación a sistemas resolutores	Escalamiento de los *tickets* automatizables a los correspondientes sistemas resolutores.
RF-4	Lectura de notificaciones	Recogida y lectura de las respuestas generadas por los sistemas resolutores
RF-5	Cierre de *tickets*	Escalamiento hacia el centro de gestión asignado y cierre del *ticket*.
RF-6	Derivación a fuerza humana	Traslado de los *tickets* no automatizables a un procesamiento manual.
RF-7	Monitorización de la cola de automatizaciones de la herramienta de *ticketing* externa	Lectura y revisión de la cola para comprobar que no contenga *tickets* atascados sin procesar.

Tabla 5.1: Requisitos funcionales del proyecto

Código	Nombre	Descripción
RNF-1	Alta disponibilidad	El sistema debe ofrecer servicio en todo momento.
RNF-2	Tolerancia a fallos	El sistema debe ser capaz de recuperarse de los fallos sin pérdida de datos.
RNF-3	Seguridad	El sistema debe garantizar la seguridad en los envíos y lecturas de mensajes.
RNF-4	Fiabilidad	El sistema debe proporcionar una ejecución fiable, asegurando que las decisiones referentes al flujo de mensajería son las correctas.
RNF-5	Escalabilidad	El sistema debe estar preparado para una alta carga de trabajo y posibles expansiones futuras.
RNF-6	Integridad de datos	El sistema debe asegurar la correctitud y completitud de los datos almacenados en base de datos.

Tabla 5.2: Requisitos no funcionales del proyecto

5.2 Diseño

En esta sección se expondrá la arquitectura general del sistema explicando como se relacionan sus componentes. También se abordarán las decisiones de diseño referentes al sistema de colas empleado.

5.2.1 Arquitectura general

En el diseño del módulo de automatización se diferencian 2 mecanismos clave:

- Un **módulo de reglas**, para decidir si una solicitud es automatizable o no y que acciones se llevan a cabo en cada caso. Este, a su vez, está divido en 3 componentes: el módulo de recepción de mensajes y notificaciones (*Listener*); el módulo de orquestación de flujo y decisión de estrategias de resolución (*Orchestrator*); y el de modificación y traslado de los *tickets* (*Worker*).
- Un **sistema de colas**, para notificar a los sistemas externos que tienen acciones que realizar y para que estos informen al sistema cuando la solicitud ha sido realizada, reflejando si fue con éxito o no.

El *Listener* será el encargado de comunicarse con el *topic* de eventos de entrada y con la cola de notificaciones. Su salida será la entrada del *Orchestrator*, que delegará en el *Worker* la ejecución de ciertas operaciones. Este último será el responsable de enviar los *tickets* a las colas de los sistemas resolutores. Así, tenemos dos módulos (*Listener* y *Worker*) totalmente independientes entre sí, conectados por un tercero (*Orchestrator*) que actúa como enlace entre ellos. Esto puede apreciarse en el esquema de la Figura 5.1.

Cabe mencionar que el *Worker* realizará sus operaciones sobre los *tickets* empleando servicios web de otros módulos ya existentes en el proyecto AXUDOT.

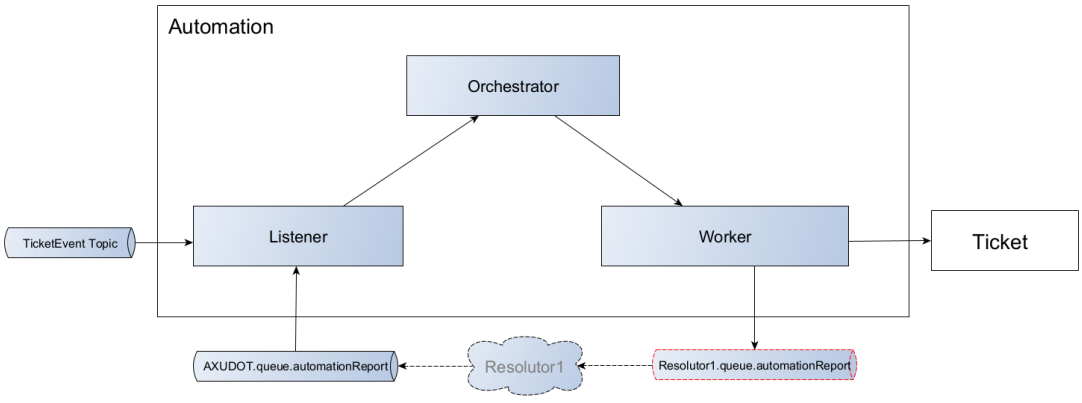


Figura 5.1: Estructura general del módulo

5.2.2 Sistema de colas

Para realizar la comunicación con los sistemas resolutores se podrían haber empleado otros métodos (como servicios web, por ejemplo), pero se ha optado por un sistema de colas debido a que proporciona las siguientes ventajas:

- **Bajo acoplamiento y tolerancia a fallos**: Aunque un sistema este caído, la solicitud quedará registrada en la cola, permitiendo así que el sistema externo la procese cuando se recupere.
- **Escalabilidad**: La integración de un nuevo sistema solo supondría un cambio en la configuración de los automatismos de nuestro módulo, ya que es el sistema externo el que implementa toda la lógica necesaria. Esto también supone que el sistema externo tiene total libertad en la manera de implementar la logica de automatización, haciéndolo así independiente de nuestro módulo.
- **Sin impacto en el modelo de servicio actual**: La solución no tiene impacto en los grupos de soporte ya que el tratamiento de las solicitudes no automatizables o de aquellas en las cuales el automatismo falle será el mismo que hasta ahora.

El sistema (ver Figura 5.2) quedaría formado por un ***topic* de entrada** que repartirá los *tickets* creados desde *frontend*, al que se tendrá que suscribir nuestro módulo de automatización. Se opta por el *topic* pensando en facilitar la escalabilidad en el futuro, ya que en caso de que otro módulo distinto también necesite recibir estos mensajes, bastará con que se suscriba. Una vez pocesados, y en caso de que sean automatizables, pasarán a la **cola de automatizaciones** propia de AXUDOT (en la herramienta externa de *ticketing*) para evitar la interacción de los agentes. Inmediatamente después, se envía a la **cola de solicitudes** específica del sistema resolutor encargado de procesarlo y, finalmente, notificar el resultado en **la cola de eventos** que leerá nuestro módulo. En caso de éxito, se retirará el *ticket* de la cola de automatizaciones; y en caso de fallo, además, se escalará a la **cola de soporte humano** (también en la herramienta externa), para que sea procesado manualmente.

5.2.3 Sistemas resolutores

En el momento de la realización de este proyecto, el módulo de automatismos dispone de tres sistemas resolutores diferentes. Cada uno de ellos se corresponde con un organismo, por lo que, por motivos de privacidad, no se citarán sus nombres reales.

Su labor será la de implementar lo siguiente:

- Un método para leer la cola de solicitudes y procesar mensajes con el formato proporcionado desde el módulo, el cual contendrá una serie de campos comunes y otros específicos para cada sistema.

```
{
    "requestId": 000000,
    "detail": {
        "beneficiary": {
            ...
        },
        "approver" : {
            ...
        },
        "requestType" : "ADD/MODIFY/DELETE",
        ...
    }
}

```

Fragmento de código 5.1: Formato de mensaje de entrada

- Un método para escribir en la cola de eventos la notificación oportuna cuando termine el procesamiento de una solicitud. El formato de este mensaje de salida será estándar para todos los sistemas.
- Un tratamiento adecuado de excepciones para que en caso de que el procesamiento falle, pueda informarse correctamente de la causa en la cola de eventos.

```
{
    "requestId": 000000,
    "completedSuccessfully": "true/false",
    "errorMessage": "En caso de error, aqui iría el mensaje"
}

```

Fragmento de código 5.2: Formato de mensaje de salida

En el futuro, se podrían incorporar nuevos sistemas resolutores al módulo. Para ello, dicho sistema deberá comunicar el nombre que tendrá su cola de solicitudes, y acordar el formato de mensaje que se empleará para escribir en dicha cola. Finalmente, se le concederán permisos de escritura sobre la cola de eventos y se le facilitará el formato de mensaje de la misma.

5.3 Casos de uso

A continuación, se muestran los dos casos de uso principales del sistema, detallando su flujo y acompañándolos de su correspondiente diagrama de secuencia.

Solicitud automática resuelta con éxito

- Este caso de uso, reflejado en la Figura 5.3, muestra el flujo de creación de un ticket que se resuelve correctamente de manera automática.
- Tras el paso de aprobación se comprueba si la solicitud es automatizable (en el futuro se contempla la posibilidad de que la aprobación sea automática).
- En caso de tener resolución automática, se escala el ticket a una cola especial para evitar la interacción de los agentes mientras éste se procesa automáticamente.
- El ticket se deja en la cola de procesado automático, esperando su resolución.
- Cuando el sistema resolutor procesa la solicitud deja un mensaje en la cola de eventos, de donde el sistema lee el mensaje y se encarga de cerrar el ticket con una nota, indicando que se ha resuelto automáticamente.

Solicitud automática con resolución fallida

- Este segundo caso de uso (Figura 5.4) representa un flujo en el que la solicitud no puede ser resuelta por el sistema resolutor.
- El flujo es idéntico al del primer caso, excepto que en el paso final, el sistema resolutor indica en la cola de eventos que no pudo procesar la solicitud, incluyendo un mensaje de error.
- Al comprobar que la solicitud no se pudo resolver, el sistema incluye una nota indicando el error y escala el ticket a la cola del CAU.
- El CAU se encarga de tratar el ticket de manera manual.

5.4 Modelo de Datos

En esta sección se analizarán las tablas accedidas por el sistema, explicando su función y cada una de sus columnas.

- Automatism[1]: Tabla que almacena la información de las estrategias de automatismo disponibles y sus aplicaciones.

[1] Esta será la única tabla accedida directamente desde el módulo de automatismos. El resto de tablas son manipuladas por otros módulos de AXUDOT accedidos mediante servicios web. Se mencionan y detallan ya que es necesario conocer su estructura para la correcta implementación de la automatización y su entendimiento y gestión.

  - request_code: Indica el código de las peticiones que podrán hacer uso del mecanismo de automatización relacionado.
  - strategy: Especifica el tipo de automatización.
  - available: Se emplea un valor entero (0 o 1) para reflejar la disponibilidad del automatismo.
  - queue_name: Almacena el nombre de la cola a la que se enviará la petición.

- Ticket: Tabla que gestiona la información de los tickets a procesar.
  - tn: Número del ticket.
  - title: Título del ticket
  - queue_id: Identificador de la cola a la que pertenece actualmente.
  - type_id: Identificador del tipo al que pertenece.
  - service_id: Identificador del servicio al que pertenece.
  - ticket_state_id: Identificador del estado en el que se encuentra.
  - customer_user_id: Nombre de usuario del solicitante.
  - create_time: Fecha de creación del ticket.
  - change_time: Fecha de modificación del ticket.

- Queue: Tabla que contiene los nombres de las distintas colas disponibles.
  - name: Nombre de la cola.

- Ticket_type: Tabla que guarda los distintos tipos de ticket.
  - name: Nombre del tipo.

- Service: Tabla que almacena los distintos servicios disponibles.
  - name: Nombre del tipo.
  - valid_id: Identificador que especifica el nivel de validez del servicio.

- Ticket_state: Tabla que proporciona los diferentes estados posibles.
  - name: Nombre del estado.

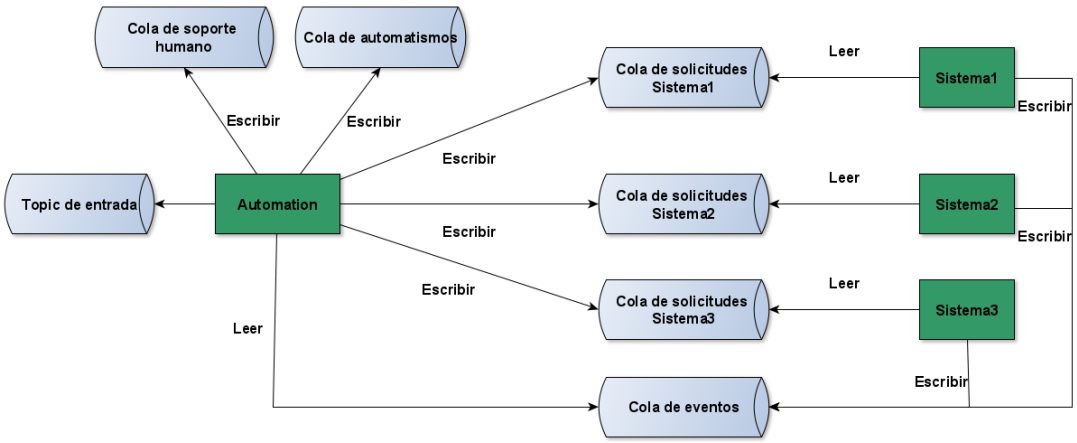


Figura 5.2: Diseño del sistema de colas

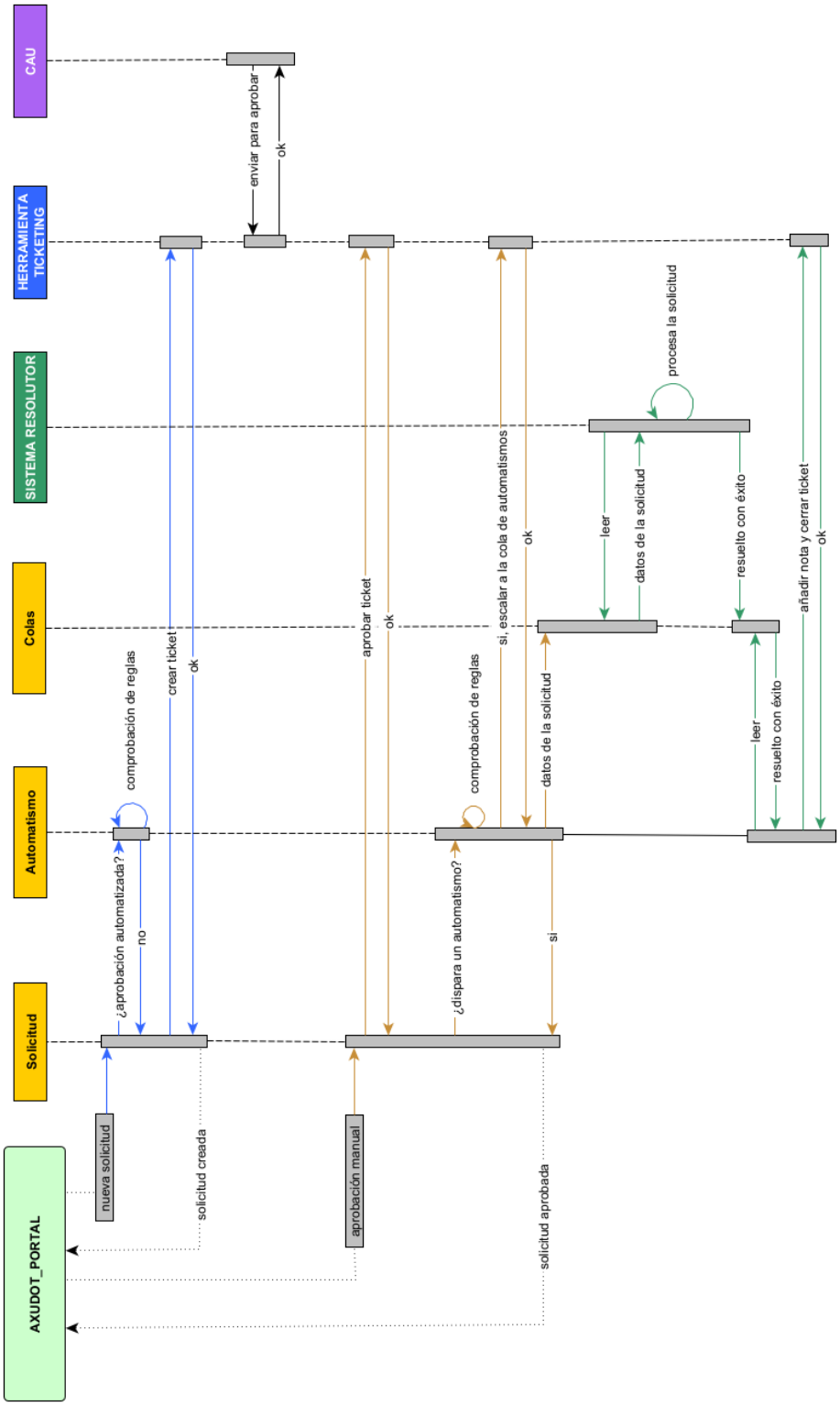


Figura 5.3: Diagrama de secuencia del caso de uso *Solicitud automática resuelta con éxito*

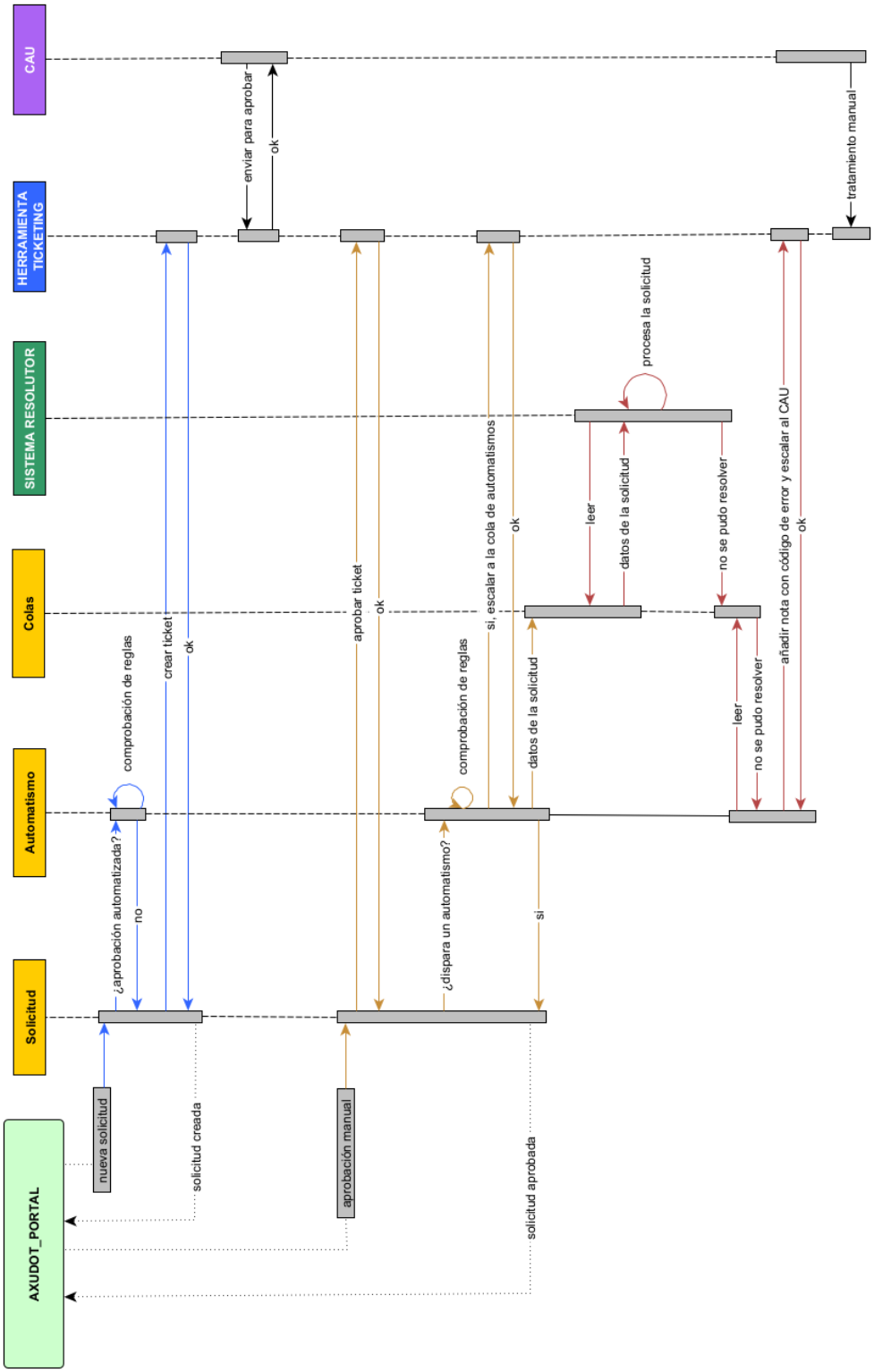


Figura 5.4: Diagrama de secuencia del caso de uso *Solicitud automática con resolución fallida*

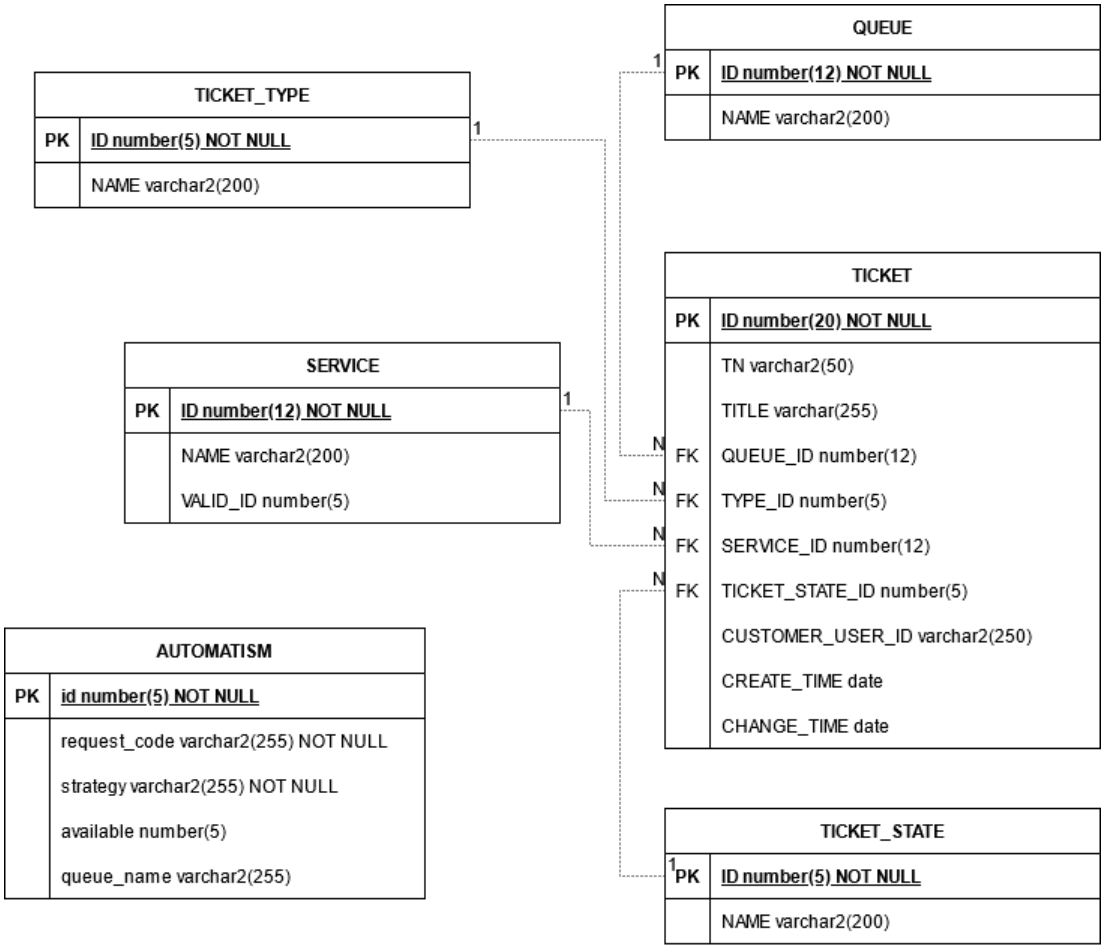


Figura 5.5: Modelo de Datos

Capítulo 6

Desarrollo

A lo largo de este capítulo se analizará con detalle todo el proceso seguido para el desarrollo de este proyecto. Para ello, se tratan con profundidad los módulos principales del proyecto, junto a su diseño e implementación.

6.1 Listener

En este apartado se recogen todos los aspectos relacionados con el módulo *Listener*, encargado de recoger los mensajes que se reciben desde el *topic* entrada y orquestarlos hacia su correspondiente resolución; así como de leer la cola de notificaciones.

6.1.1 Análisis

Este componente funciona como enlace con los sistemas origen y comprueba las respuestas obtenidas por parte de los resolutores. Para ello, dispone de dos *listeners* especializados y diferenciados:

- `TicketEventMessageListener`: encargado de recoger los *tickets* recibidos desde los sistemas origen, convirtiéndolos a su formato particular y delegando su manipulación al método correspondiente en el *orchestrator*.

- `AutomationReportMessageListener`: su trabajo es leer la cola de notificaciones en la que escriben los sistemas resolutores. Adapta estos mensajes a su formato y los envía al *orchestrator* para su procesamiento.

Un *listener* es un objeto que actúa como un controlador asíncrono de eventos para mensajes. Implementa la interfaz `MessageListener`, la cual contiene un único método (`onMessage`) en el que se definen las acciones que se llevarán a cabo cuando se reciba un mensaje. Mediante

el método `setMessageListener`, especificamos cuál será el consumidor concreto que emplearemos. Cuando comienza la entrega de mensajes, JMS invoca automáticamente al método `onMessage` del *listener* con cada envío.

El mismo *listener* puede obtener mensajes tanto de un *topic* como de una cola, dependiendo de la intención con la que el consumidor fue creado. Sin embargo, un *listener* suele esperar un tipo y formato concreto de mensaje. El método `onMessage` debe controlar todas las excepciones, no debe lanzar ninguna.

6.1.2 Diseño e implementación

Arquitectura

Este módulo no tendrá que realizar ningún acceso a datos ni exponer ningún tipo de servicio web; se limitará a interactuar con las distintas colas haciendo uso de la interfaz `MessageListener` y a enviar los mensajes hacia el *Orchestrator*, por lo que no requiere seguir ningún patrón concreto.

Se opta por una división simple entre *api* y *core*, en la que las interfaces `MessageListener` y `MessageConverter` encajarían en la capa *api* pero, al estar incluidas en las librerías, dicha capa queda inutilizada, resultando en la arquitectura que se puede observar en la Figura 6.1.

Paquetes

El componente presenta los siguientes paquetes:

- **axudot.automation.listener.core.jms.configuration**: Contiene la clase de configuración de los dos *listeners* empleados por el componente.
- **axudot.automation.listener.core.jms.dto**: Este paquete agrupa los objetos empleados para representar las entidades.
- **axudot.automation.listener.core.jms.converter**: Contiene las clases encargadas de convertir los Data Transfer Object (DTO) a objetos `Message` y viceversa.
- **axudot.automation.listener.core.jms.listener**: En este paquete se encuentran las implementaciones del API `MessageListener`.

Clases y funcionalidades

En este apartado se comentarán las principales clases del módulo junto a sus funciones dentro del mismo:

- **AutomationListenerJmsConfiguration**: Clase de configuración de los *listeners*, en la que se utiliza la anotación `@Bean` para inyectar las dependencias necesarias para cada uno de los *listeners*. A continuación en el código 6.1 se pueden ver los *beans* correspondientes a `TicketEventMessageListener` y `AutomationReportMessageListener`. Se puede apreciar como al configurar la conexión del primero se invoca el método `setPubSubDomain` pasándole *true* como parámetro para indicar que se trata de una suscripción a un *topic*.

```
1   @Bean
2   public MessageListenerContainer
    automationTicketEventMessageListenerContainer(
    TicketEventMessageListener messageListener, ConnectionFactory
    connectionFactory, PropertyManager propertyManager) {
3
4       DefaultMessageListenerContainer messageListenerContainer =
      new DefaultMessageListenerContainer();
5
    messageListenerContainer.setConnectionFactory(connectionFactory);
6
    messageListenerContainer.setMessageListener(messageListener);
7       messageListenerContainer.setDestination(new ActiveMQTopic(
    propertyManager.getProperty( "ticket.event.topic.name" )));
8       messageListenerContainer.setPubSubDomain(true);
9       messageListenerContainer.setSubscriptionShared(true);
10
11      return messageListenerContainer;
12  }
13
14  @Bean
15  public MessageListenerContainer
    automationReportMessageListenerContainer(
    AutomationReportMessageListener messageListener,
    ConnectionFactory connectionFactory, PropertyManager
    propertyManager) {
16
17      DefaultMessageListenerContainer messageListenerContainer =
    new DefaultMessageListenerContainer();
18
    messageListenerContainer.setConnectionFactory(connectionFactory);
19
    messageListenerContainer.setMessageListener(messageListener);
20      messageListenerContainer.setDestinationName(
    propertyManager.getProperty("automation.report.queue.name"));
21
22      return messageListenerContainer;
```

```
23      }
24
```

Fragmento de código 6.1: Clase de configuración AutomationListenerJmsConfiguration

- **DTOs**: Objetos empleados para manejar la información de los mensajes. Al ser recibidos, se procesan con los *converter* y se obtienen estos objetos como resultado. A continuación se muestra el DTO de las notificaciones de los sistemas resolutores (fragmento 6.2) y el de los mensajes recibidos desde los sistemas origen (fragmento 6.3).

```
1     public class AutomationReportMessage implements Serializable {
2         ...
3
4         private Long requestId;
5         private boolean completedSuccessfully;
6         private String errorMessage;
7
8         public AutomationReportMessage() {
9         }
10
11        ...
12
13        public Long getRequestId() {
14            return this.requestId;
15        }
16
17        public void setRequestId(Long requestId) {
18            this.requestId = requestId;
19        }
20
21        ...
22    }
23
```

Fragmento de código 6.2: DTO AutomationReportMessage

```
1  public class TicketEventMessage implements Serializable {
2      ...
3
4      private Long ticketId;
5      private TicketState state;
6      private TicketEventType type;
7
8      public TicketEventMessage() {
9      }
10
```

```
    ...

    public Long getTicketId() {
        return this.ticketId;
    }

    public void setTicketId(Long ticketId) {
        this.ticketId = ticketId;
    }

    ...
}

```

Fragmento de código 6.3: DTO TicketEventMessage

- ***Listeners***: Como se explicó con anterioridad, son las clases encargadas de detectar los mensajes recibidos (gracias a los *beans* previamente inyectados), convertirlos a los DTOs correspondientes y trasladarlos al módulo *Orchestrator* para su procesamiento. En los fragmentos siguientes se muestra la implementación de los métodos `onMessage` en los dos *listeners* presentes en el sistema.

```
public class AutomationReportMessageListener implements MessageListener {
    ...

    public void onMessage(Message message) {
        try {
            AutomationReportMessage msg = (AutomationReportMessage) this.automationReportMessageConverter.fromMessage(message);
            this.orchestrator.processAutomationReport(new AutomationReport(msg.getRequestId(), msg.isCompletedSuccessfully(), msg.getErrorMessage()));
        } catch (JMSException e) {
            LOGGER.error("Error receiving message", e);
        }
    }
}

```

Fragmento de código 6.4: Clase AutomationReportMessageListener

```
public class TicketEventMessageListener implements MessageListener {
    ...

    public void onMessage(Message message) {
```

```
5           try {
6               TicketEventMessage msg = (TicketEventMessage)
        this.ticketEventMessageConverter.fromMessage(message);
7               this.orchestrator.processTicketEvent(new TicketEvent(
8                       msg.getTicketId(),
9                       TicketState.valueOf(msg.getState().name()),
10                      TicketEventType.valueOf(msg.getType().name())));
11          } catch (JMSException e) {
12              LOGGER.error("Error receiving message", e);
13          } catch (InputValidationException e) {
14              LOGGER.error("Error processing message", e);
15          }
16      }
17  }
18
```

Fragmento de código 6.5: Clase TicketEventMessageListener

6.2 Orchestrator

En este apartado se recogen todos los aspectos relacionados con el módulo *Orchestrator*, encargado de determinar si un *ticket* es o no automatizable, y que estrategia seguirá en caso de que lo sea. Accederá a la base de datos para consultar los automatismos disponibles y delegará ciertas tareas en el módulo *Worker*.

6.2.1 Análisis

Esta unidad actúa como enlace entre los módulos *Listener*, del que recibe los mensajes; y *Worker*, en el que delega la ejecución de ciertas operaciones. Es el mayor componente del sistema, y llevará a cabo la labor de orquestar el flujo de cada mensaje y derivar el trabajo a un canal de fuerza humana, en caso de ser necesario.

Para conseguirlo, dispone de un servicio en el que se procesa cada *ticket* recibido, comprobando si dispone de un automatismo asociado, y evaluando cada caso particular. Dependiendo del resultado de dicha evaluación, delega la manipulación del *ticket* en la estrategia adecuada para que esta lo cambie de cola y, si procede, lo cierre.

También proporciona un API REpresentational State Transfer (REST)[16] accesible desde servicios web, el cual incluye un servicio de monitorización del estado de la cola de entrada del sistema; así como un repositorio para consultar y gestionar la base de datos de automatismos.

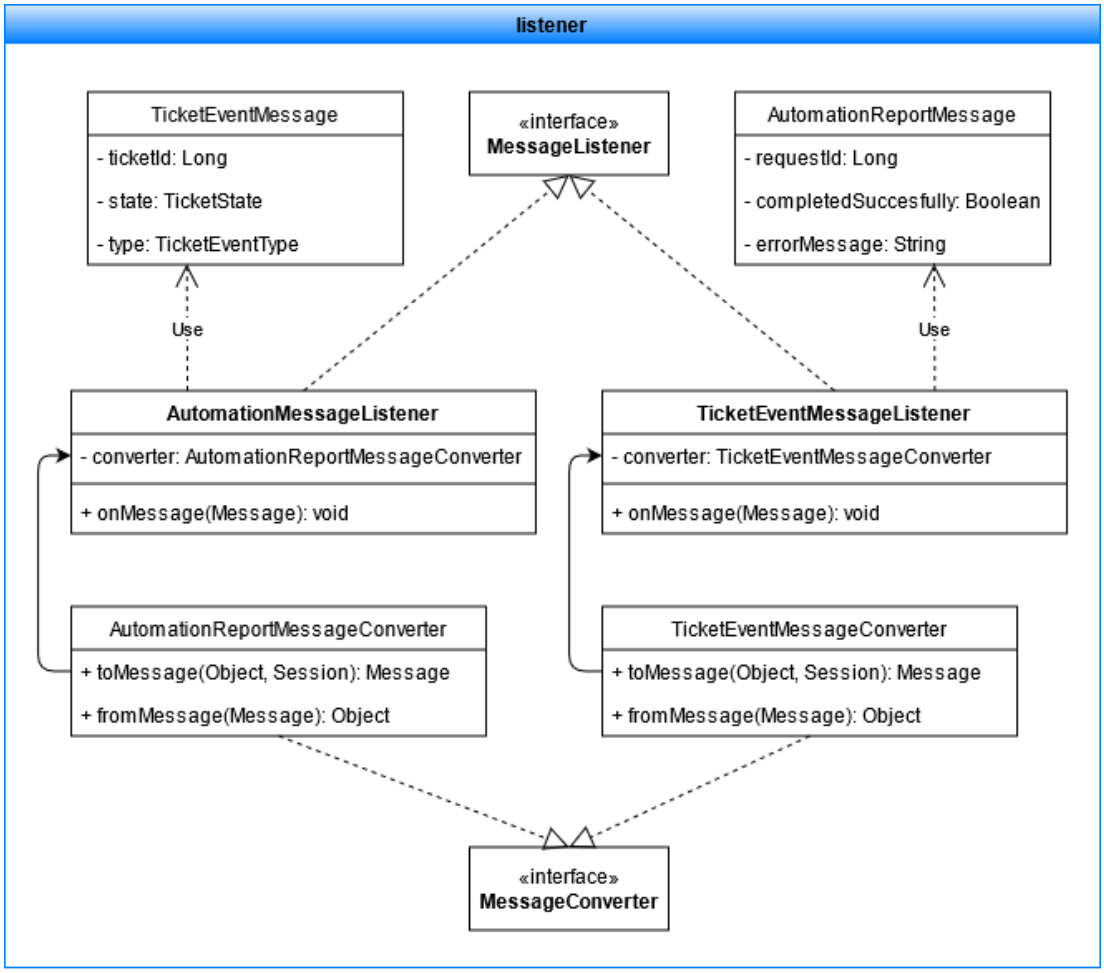


Figura 6.1: Diagrama de clases del módulo Listener

6.2.2 Diseño e implementación

Arquitectura

El módulo *Orchestrator*, como se ha comentado previamente, presenta un API REST, un servicio y un repositorio de acceso a datos. Teniendo esto en cuenta, la solución más adecuada es la de seguir una arquitectura por capas (sección 4.1), con cada una de ellas dividida a su vez en *api* y *core*. En este caso, el número de capas empleadas es tres:

- *Presentation*: Expone el API REST junto a su implementación.
- *Service*: Contiene dos servicios (el principal del módulo y el empleado por el API REST) y las distintas estrategias tanto de automatización como de gestión de notificaciones.
- *Persistence*: Proporciona el repositorio de acceso a base de datos para la gestión de los automatismos.

Esta distribución puede apreciarse con mayor detalle en la Figura 6.2.

Paquetes

Las clases están distribuidas en los siguientes paquetes:

- **axudot.automation.orchestrator.configuration**: Contiene la clase de configuración del componente.
- **axudot.automation.orchestrator.persistence.api**: Empaqueta la interfaz del repositorio, así como las entidades y DTOs, repartidas en sus respectivos sub-paquetes (entity y dto).
- **axudot.automation.orchestrator.persistence.core**: Formado por la clase de implementación del repositorio y la interfaz JPA empleada por el mismo.
- **axudot.automation.orchestrator.service.api**: Incluye las interfaces de los servicios, junto con los DTOs empleados por estos (nuevamente, en sus respectivos paquetes dto.automation y dto.ticket).
- **axudot.automation.orchestrator.service.core**: En este paquete se encuentran las implementaciones de los servicios y las clases encargadas de definir las distintas estrategias. Se divide en varios contenedores:
  - **core._util**: Contiene diversas clases de utilidad para el módulo.
  - **core.configuration**: Aquí se recogen las configuraciones del servidor de correo y de las estrategias de notificaciones exitosas.

  - **core.procedure**: Alberga la clase encargada de ejecutar el procedimiento establecido cuando la automatización falla.
  - **core.strategy**: Reúne las interfaces para las estrategias tanto de automatizaciones como de notificaciones exitosas. Además, presenta otros sub-paquetes:
    * **strategy.automation**: Contiene la clase encargada de la estrategia de automatización para los tickets aprobados.
    * **strategy.resolutorN**: Siendo N el número del sistema resolutor, estos paquetes recogen las estrategias específicas de notificaciones exitosas para cada sistema en concreto.

- **axudot.automation.orchestrator.presentation.api**: Formado por la interfaz del API REST que expone los servicios web.

- **axudot.automation.orchestrator.presentation.core**: Incluye la implementación de la interfaz del API REST anteriormente mencionado.

Clases y funcionalidades

En este apartado se comentarán las principales clases del módulo junto a sus funciones dentro del mismo:

- **AutomatismWebService**: API REST que proporciona las operaciones `setAvailable`, que activa o desactiva un automatismo en función de lo que se le indique; y `checkAutomatismQueue`, que a partir de los minutos proporcionados, devuelve la lista de *tickets* que tienen una fecha de modificación anterior a dichos minutos.

```
1  @Path("/automation")
2  public interface AutomatismWebService {
3
4      @PUT
5      @Path("/{requestCode}")
6      void setAvailable(@PathParam("requestCode") String requestCode,
       @QueryParam("available") Boolean isAvailable) throws
       InputValidationException;
7
8      @GET
9      @Path("/check/{minutes}")
10     @Produces(MediaType.APPLICATION_JSON)
11     List<TicketQueueDto> checkAutomatismQueue(@PathParam("minutes")
       Long minutes);
12 }
13
```

Fragmento de código 6.6: API REST AutomatismWebService

- **AutomatismService**: Servicio responsable de ejecutar las operaciones presentadas en el API REST. En él se lleva a cabo la monitorización de la cola de automatismos, cuya implementación se puede observar en el fragmento 6.7. Además, junto con la lectura de los *tickets*, se realiza una notificación vía correo electrónico mediante el uso del API de JavaMail (sección 4.4).

```
@Service
public class DefaultAutomatismService implements AutomatismService {

    ...

    @Override
    public List<TicketQueueDto> checkAutomatismQueue(Long minutes) {
        List<TicketQueueDto> tickets =
    this.ticketWorker.getTicketsFromQueue(
    TicketQueue.AXUDOT_AUTOMATION.getName());
        List<TicketQueueDto> ticketList = new ArrayList<>();
        Date filter = getFilterDate(minutes);

        for (TicketQueueDto ticket : tickets) {
            if (ticket.getLastModificationDate().before(filter)) {
                ticketList.add(ticket);
            }
        };

        sendMail(ticketList);

        return ticketList;
    }

    ...

    private void sendMail(List<TicketQueueDto> list){

        String message = ... ;

        Message msg = new MimeMessage(mailSession);

        try {
            // from
            msg.setFrom(new
    InternetAddress(propertyManager.getProperty(
```

```
    "email.automatism.sender")));

            // to
            msg.setRecipients(Message.RecipientType.TO,
    InternetAddress.parse(propertyManager.getProperty(
    "email.automatism.receiver"), false));

            // subject
            msg.setSubject("Tickets pendentes na cola de
    automatismos");

            // content
            msg.setContent(message, "text/html; charset=utf-8");

            msg.setSentDate(new Date());

            // send
            Transport.send(msg);
        } catch (MessagingException e) {
            LOGGER.error("Error sending email.");
        }

    }

```

Fragmento de código 6.7: Implementación del método checkAutomatismQueue

```
@Configuration
public class MailConfiguration {

    @Autowired
    private PropertyManager propertyManager;

    @Bean
    public Session mailSession(){
        Properties prop = new Properties();
        prop.setProperty("mail.smtp.host", "hostname");
        prop.setProperty("mail.smtp.auth", "true");
        prop.setProperty("mail.smtp.starttls.enable", "true");
        prop.setProperty("mail.smtp.port", "portnumber");
        prop.setProperty("mail.transport.protocol", "smtp");

        return Session.getInstance(prop,
                new javax.mail.Authenticator() {

                    //override the getPasswordAuthentication method
```

```
                    protected PasswordAuthentication
                    getPasswordAuthentication() {
                        return new PasswordAuthentication(
    propertyManager.getProperty("auth.app.username"),
    propertyManager.getEncryptedProperty("auth.app.password"));
                    }
                });
    }
}

```

Fragmento de código 6.8: Configuración de la sesión JavaMail

- **Orchestrator**: Servicio que implementa las funciones de procesamiento de los *tickets* y las notificaciones recogidas por el módulo Listener. Presenta dos métodos públicos:
  - `processTicketEvent`: Método asociado a los *tickets* recibidos desde el *topic* de entrada. Comprueba si es automatizable y, en caso de serlo, se encarga de invocar la ejecución de la estrategia correspondiente.

```
public Boolean processTicketEvent(TicketEvent event) {
        TicketSummaryDto summary =
    this.ticketWorker.getSummary(event.getTicketId());
        Automatism automatism;
        try {
            automatism =
    this.automatismRepository.getAutomatism(summary.getRequestCode());
            String strategyName = automatism.getStrategyName();
            if (strategyName != null) {
                if (automatism.getAvailable() == 1) {
                    AutomationStrategy strategy =
    (AutomationStrategy) this.appContext.getBean(strategyName);
                    if (strategy.isAutomatable(event)) {
                        strategy.execute(event,
    automatism.getQueueName());
                        return true;
                    } else {
                        LOGGER.debug("The rule doesn't apply");
                    }
                } else {
                    LOGGER.info("Automation strategy {} is not
    available for <{}>", strategyName, automatism.getRequestCode());
                }
            } else {
                LOGGER.debug("No automation strategy found for
    requests with code {}", summary.getRequestCode());
```

```
            }
            return false;
        } catch (InputValidationException e) {
            return false;
        }
    }

```

Fragmento de código 6.9: Implementación del método `processTicketEvent`

- `processAutomationReport`: Método asociado a las notificaciones recibidas desde los sistemas resolutores. Comprueba si la resolución del *ticket* fue exitosa o no. En caso de serlo, ejecuta la estartegia asociada (o la estrategia por defecto en caso de no existir ninguna). En caso contrario, invoca la ejecución del proceso de error en la resolución.

```
public Boolean processAutomationReport(AutomationReport report) {
        if (report.isCompletedSuccessfully()) {
            TicketSummaryDto summary =
    this.ticketWorker.getSummary(report.getRequestId());
            SuccessfulAutomationReportStrategy strategy =
    this.successfulAutomationReportStrategyMap.get(
    summary.getRequestCode());
            if (strategy == null) {

    this.defaultSuccessfulAutomationReportStrategy.execute(
    report.getRequestId());
            } else {
                strategy.execute(report.getRequestId());
            }
            return true;
        } else {
            this.failedAutomationReportProcedure.execute(
    report.getRequestId(), report.getErrorMessage());
            return false;
        }
    }

```

Fragmento de código 6.10: Implementación del método `processAutomationReport`

- **AutomationStrategy**: Interfaz que presenta los métodos `isAutomatable`, que devuelve *true* si el *ticket* puede ser automatizado y *false* en caso contrario; y `execute`, que escribe una nota en el *ticket* reflejando su escalamiento debido a la automatización,

lo traslada a la cola de automatismos de AXUDOT y, finalmente, lo envía a la cola del sistema resolutor correspondiente.

```
    public boolean isAutomatable(TicketEvent event) {
            return event.getState().equals(TicketState.APPROVED) &&
    event.getType().equals(TicketEventType.UPDATE) &&
    this.isAnCreateRequest(event.getTicketId());
        }

```

Fragmento de código 6.11: Implementación del método `isAutomatable` de la clase DefaultAutomationStrategy

```
public void execute(TicketEvent event, String queueName) {
        this.writeInformativeNoteOnTicket(event.getTicketId());
        this.ticketWorker.changeQueue(event.getTicketId(),
    TicketQueue.AXUDOT_AUTOMATION);

        this.getDetailAndSendItToQueue(event.getTicketId(),
    queueName);
    }

```

Fragmento de código 6.12: Implementación del método `execute` de la clase DefaultAutomationStrategy

- **SuccessfulAutomationReportStrategy**: Interfaz que proporciona el método `execute`, que se encarga de añadir una nota que indique la resolución automática exitosa, devuelva el *ticket* a la cola correspondiente y, por último, lo cierre. Se dispone de una implementación por defecto y de otra para cada uno de los sistemas resolutores. La única diferencia entre ellas será la escritura en los logs, pero se realiza esta distinción de cara a modificaciones futuras, con intención de facilitar la escalabilidad.

```
public void execute(Long requestId) {
        this.ticketWorker.addNote(requestId, new NoteDto(
                        "Resolución automática",
                        "Ticket resolto de maneira automática"),
                "note-internal");
        this.queueManager.changeQueueToAssignedUsc(requestId);
        this.ticketWorker.close(requestId);
    }

```

Fragmento de código 6.13: Implementación del método `execute` de la clase DefaultSuccesfulAutomationReportStrategy

- **FailedAutomationReportProcedure**: Clase que ejecuta el procedimiento requerido ante el fallo de la automatización: incluye una nota en el *ticket* informando del fallo y lo escala hacia trabajo de fuerza humana.

```
public void execute(Long requestId, String errorMessage) {
        this.ticketWorker.addNote(requestId, new NoteDto(
                        "Fallo na resolución automatizada do ticket",
                        errorMessage),
                "note-internal");
        this.queueManager.changeQueueToDigitalProcedures(requestId);
    }

```

Fragmento de código 6.14: Implementación del método `execute` de la clase FailedAutomationReportProcedure

- **AutomatismRepository**: Interfaz que contiene los métodos necesarios para trabajar con la tabla de Automatism en base de datos. `getAutomatism` devuelve la entidad asociada al código proporcionado, mientras que `updateAutomatism` actualiza el valor de disponibilidad del automatismo en función del indicado.

```
@Repository
public class DefaultAutomatismRepository implements AutomatismRepository {
    ...

    @Override
    public Automatism getAutomatism(String requestCode) throws InputValidationException {
        AutomatismEntity entity = this.retrieveAutomatism(requestCode, this.automatismJpaRepository.findByRequestCode(requestCode));
        if (entity != null) {
            return new Automatism(entity.getId(), entity.getRequestCode(), entity.getStrategyName(), entity.getQueueName(), entity.getAvailable());
        } else {
            String message = "Automatism for <" + requestCode + "> not found";
            throw new InstanceNotFoundException(EXCEPTION_SYSTEM, ErrorType.ACCESS, ExceptionLevel.NORMAL,
                    EXCEPTION_CODE, message, message);
        }
    }

```

```
17      @Override
18      public Long updateAutomatism(String requestCode, Boolean
        isAvailable) {
19          if (isAvailable == null) {
20              return null;
21          } else {
22              AutomatismEntity automatism = null;
23              try {
24                  automatism = this.retrieveAutomatism(requestCode,
        this.automatismJpaRepository.findByRequestCode(requestCode));
25                  if (automatism != null) {
26                      automatism.setAvailable(isAvailable ? 1 : 0);
27                      return automatism.getId();
28                  } else {
29                      String message = "Automatism for <" +
        requestCode + "> not found";
30                      LOGGER.error(message);
31                  }
32              } catch (InputValidationException e) {
33                  String message = "Error while recovering automatism
        for <" + requestCode + ">";
34                  LOGGER.error(message);
35              }
36              return null;
37          }
38      }
39
40      private AutomatismEntity retrieveAutomatism(String requestCode,
        List<AutomatismEntity> automatisms) throws
        InputValidationException {
41          if (automatisms == null || automatisms.isEmpty()) {
42              return null;
43          } else if (automatisms.size() > 1) {
44              String message = "Request code <" + requestCode + ">
        can't have more than 1 automatism associated";
45              throw new InputValidationException(EXCEPTION_SYSTEM,
        ErrorType.DATA_BASE, ExceptionLevel.NORMAL,
46                      EXCEPTION_CODE, message, message);
47          }
48          return automatisms.get(0);
49      }
50  }
51
```

Fragmento de código 6.15: Implementación de la interfaz AutomatismRepository

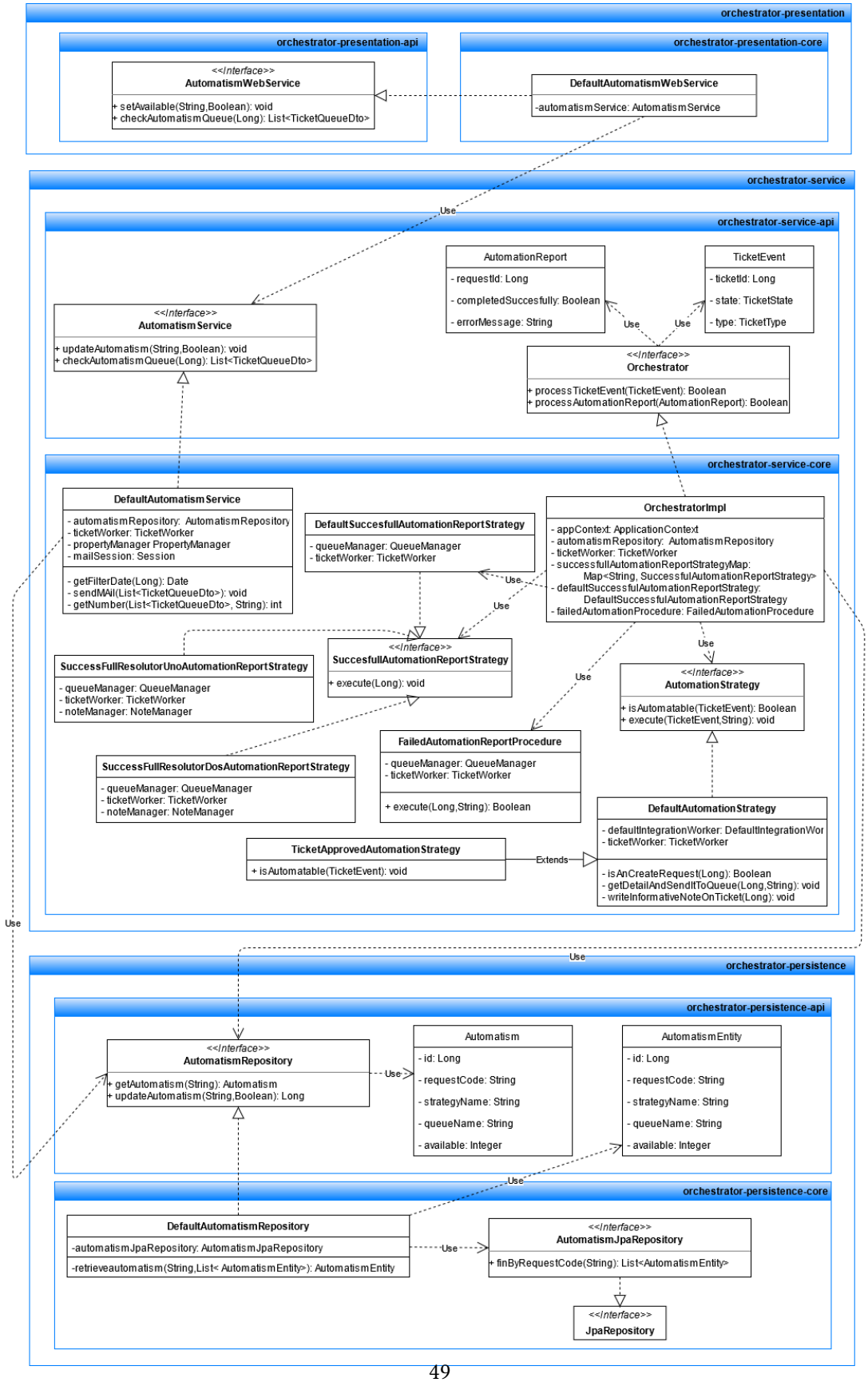


Figura 6.2: Diagrama de clases del módulo Orchestrator

6.3 Worker

En esta sección se detallan todos los aspectos relacionados con el módulo Worker, encargado de realizar los envíos de los *tickets* a las distintas colas, así como las operaciones sobre los mismos.

6.3.1 Análisis

Este componente, como su nombre indica, cumple un rol de ”trabajador”: ejecuta las operaciones que le encarga el Orchestrator. Su principal función será la de mover los *tickets* entre las diferentes colas del sistema y escribir notas informativas en los mismos, aunque, como se verá más adelante, también dispone de alguns funciones más.

Para ello, se comunica mediante servicios web con otros módulos de AXUDOT más centrados en la manipulación de los tickets y en la gestión de usuarios, por lo que se libera en gran medida de la lógica correspondiente a estas funciones. Además, hace uso de la interfaz JmsTemplate[1] de Spring, la cual proporciona métodos para crear y liberar conexiones y para convertir y enviar mensajes, entre otros.

6.3.2 Diseño e implementación

Arquitectura

Dado que este componente no realizará ninguno acceso directo a datos y no presenta más que las interfaces web necesarias para la comunicación con el resto de módulos y tres servicios, una simple división entre la capa *api* y la capa *core* será suficiente para su propósito.

Paquetes

El componente presenta los siguientes paquetes:

- **axudot.automation.worker.api.external**: Contiene el DTO representativo del mensaje recibido y la interfaz del servicio de envío.
- **axudot.automation.worker.api.ticket**: Incluye los DTOs necesarios para representar un *ticket* y el servicio que proporciona los métodos para manipularlos.
- **axudot.automation.worker.api.user**: Agrupa el DTO enumerado del centro asignado al usuario (para trasladar el ticket a su cola asociada en caso de que la automatización se haya ejecutado con éxito) y el servicio para obtener el mismo.

[1] https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/jms/core/JmsTemplate.html

- **axudot.automation.worker.core.configuration**: En este paquete se encuentran las clases de configuración de los clientes web y de JmsTemplate.
- **axudot.automation.worker.core.external**: Contiene la implementación del servicio de envío y su conversor de mensajes.
- **axudot.automation.worker.core.ticket**: Incluye la implementación del servicio de *ticket* y su cliente web.
- **axudot.automation.worker.core.user**: Incluye la implementación del servicio de obtención del Centro de Atención al Usuario (CAU) del usuario y su cliente web.

Clases y funcionalidades

En este apartado se comentarán las principales clases del módulo junto a sus funciones dentro del mismo:

- **DefaultIntegrationWorker**: Interfaz que presenta el método `send` que envía el mensaje dado a la cola indicada, utilizando para ello la interfaz JmsTemplate.

```
1  @Service
2  public class DefaultJmsIntegrationWorker implements
       DefaultIntegrationWorker {
3      ...
4      private final JmsTemplate jmsTemplate;
5      private final DefaultIntegrationMessageConverter converter;
6
7      @Autowired
8      public
       DefaultJmsIntegrationWorker(@Qualifier("defaultJmsTemplate")
       JmsTemplate jmsTemplate, DefaultIntegrationMessageConverter
       converter) {
9          ...
10     }
11
12     @Override
13     public void send(DefaultInputMessage message, String queueName)
       {
14         try {
15             this.jmsTemplate.convertAndSend(queueName,
       this.converter.convertToJsonString(message));
16         } catch (Exception e) {
17             LOGGER.error("Error sending message to the {}
       automation queue", queueName, e);
18         }
19     }
```

```
}

```

Fragmento de código 6.16: Implementación de la interfaz DefaultIntegrationWorker

- **TicketWebServiceClient**: Cliente web que sirve de enlace con los servicios web de los módulos externos, estbleciendo los paths de forma que coincidan con los de estos módulos.

```
@Path("/v2/ticket")
public interface TicketWebServiceClient {

    @GET
    @Path("/summary/{id}")
    @Produces(MediaType.APPLICATION_JSON)
    TicketSummaryDto getSummary(@PathParam("id") Long id);

    @GET
    @Path("/{id}")
    @Produces(MediaType.APPLICATION_JSON)
    TicketDetailDto getDetail(@QueryParam("requester") String
    requester, @PathParam("id") Long id);

    @PUT
    @Path("/{id}/note")
    @Consumes(MediaType.APPLICATION_JSON)
    @Produces(MediaType.APPLICATION_JSON)
    void addNote(@PathParam("id") Long id, NoteDto note);

    @PUT
    @Path("/{id}/note/{type}")
    @Consumes(MediaType.APPLICATION_JSON)
    @Produces(MediaType.APPLICATION_JSON)
    void addNoteWithType(@PathParam("id") Long id, NoteDto note,
    @PathParam("type") String type);

    @DELETE
    @Path("/{id}")
    void close(@PathParam("id") Long id);

    @PUT
    @Path("/{id}/queue")
    void changeQueue(@PathParam("id") Long id, @QueryParam("name")
    String name);

    @GET
```

```
35      @Path("search/queue/{queueName}")
36      @Produces(MediaType.APPLICATION_JSON)
37      List<TicketQueueDto>
        getTicketsFromQueue(@PathParam("queueName") String queueName);
38
39  }
40
```

Fragmento de código 6.17: Cliente web TicketWebServiceClient

- **TicketWorker**: Interfaz que recoge los métodos disponibles para la manipulación y gestión de los *tickets*. Su implementación simplemente consiste en invocar los métodos del cliente web equivalentes.

```
1  public interface TicketWorker extends Worker {
2
3      void addNote(Long ticketId, NoteDto input);
4
5      void addNote(Long ticketId, NoteDto input, String type);
6
7      void close(Long ticketId);
8
9      void changeQueue(Long ticketId, TicketQueue queue);
10
11     TicketSummaryDto getSummary(Long ticketId);
12
13     TicketDetailDto getDetail(Long ticketId);
14
15     List<TicketQueueDto> getTicketsFromQueue(String name);
16
17 }
18
```

Fragmento de código 6.18: Interfaz TicketWorker

- **UserWebServiceClient**: Cliente web que sirve de enlace con los servicios web de los módulos externos, estbleciendo los paths de forma que coincidan con los de estos módulos.

```
1  @Path("/user")
2  public interface UserWebServiceClient {
3
4      @GET
5      @Produces(MediaType.APPLICATION_JSON)
6      UserDto find(@QueryParam("username") String username);
7
```

```
}

```

Fragmento de código 6.19: Cliente web UserWebServiceClient

- **UserWorker**: Esta interfaz define el método `getAssignedUSC`, que devuelve el centro asignado al usuario dado, o, en caso de no disponer de ninguno, el centro por defecto. Para esto, emplea el cliente web para obtener el usuario.

```
@Component
public class LdapUserWorker implements UserWorker {

    ...

    @Override
    public AssignedCenter getAssignedCenter(String username) {
        final UserDto user =
    this.userWebServiceClient.find(username);
        for (AssignedCenter center :
    EnumSet.allOf(AssignedCenter.class)) {
            if (user.getCenterAssigned().equals(center.getName())) {
                return center;
            }
        }
        return AssignedCenter.CENTRAL_CENTER;
    }

}

```

Fragmento de código 6.20: Implementación de la interfaz UserWorker

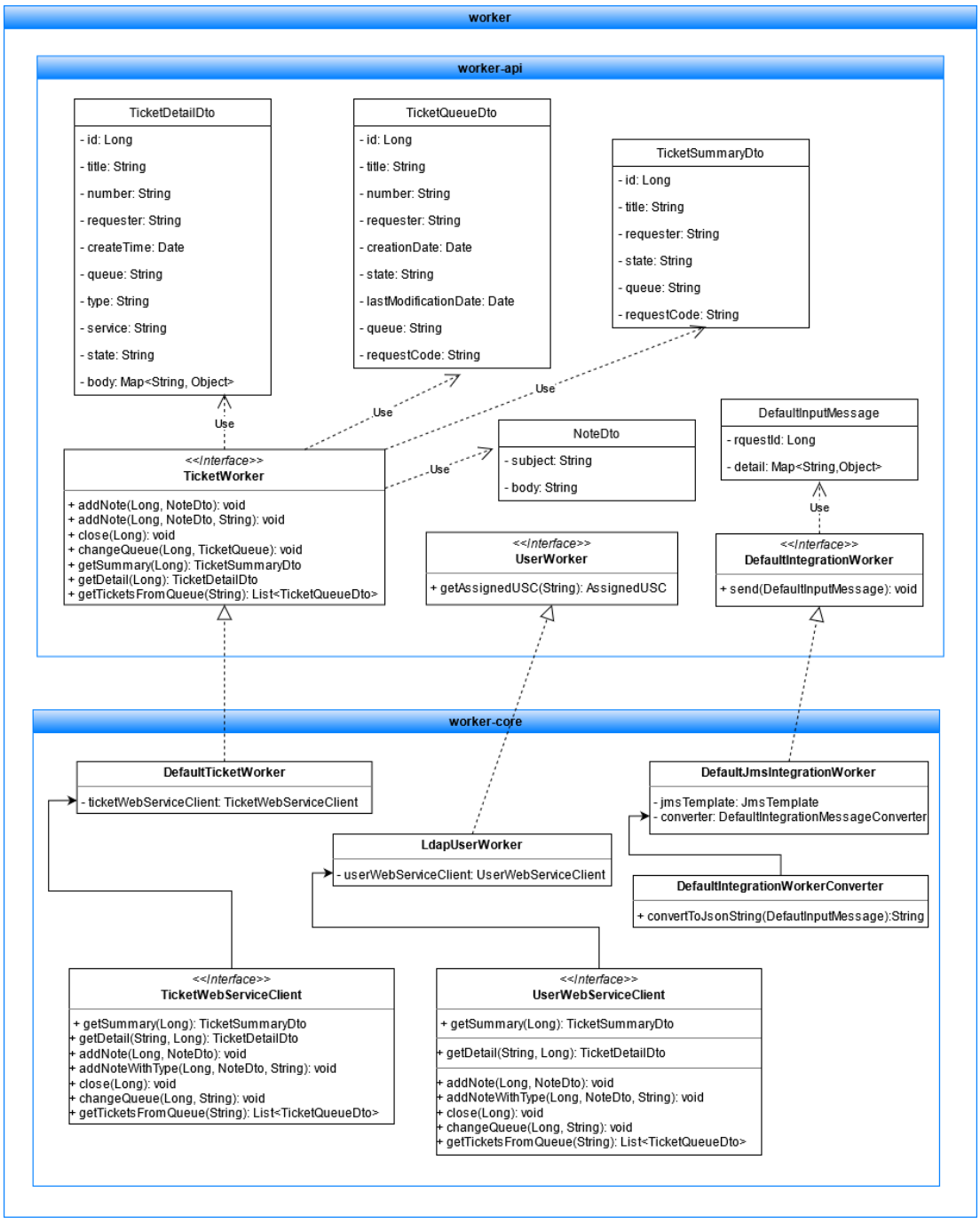


Figura 6.3: Diagrama de clases del módulo Worker

Capítulo 7

Pruebas

PARA comprobar el correcto funcionamiento del sistema desarrollado, se han realizado una serie de pruebas tanto de unidad como de integración sobre los distintos componentes del mismo, las cuales se exponen en este capítulo. Además, también se detallarán pruebas no funcionales centradas en el rendimiento y eficacia del sistema.

7.1 Pruebas de unidad

Estas pruebas se encargan de comprobar el correcto funcionamiento de fragmentos de código de forma aislada, sin que se comunique con otros componentes. Para ello, se simularán estos últimos mediante el uso de Mockito[1]. Este framework permite simular el comportamiento del resto de componentes, asegurándonos así de que nuestro código probado funciona correctamente independientemente de la implementación de los mismos.

Estas pruebas se han realizado sobre repositorio y servicios, y con ello quedan probadas de forma indirecta las entidades y los DTOs involucrados. Se estableció una meta de cobertura mínima sobre el código de un 80% con el fin de asegurar unas pruebas exhaustivas y completas, alcanzando finalmente un 89%. El porcentaje restante se corresponde con métodos tipo *setters/getters* y con algunas sentencias de lanzamiento de excepciones y/o escritura en archivos de log. Cabe destacar que no se encontraron errores importantes durante la realización de las mismas.

Para tener una referencia de la dimensión de las pruebas respecto a la implementación, se han calculado las Source Lines Of Code (SLOC) tanto de las clases de prueba como de las clases de servicios/repositorios, obteniéndose los siguientes valores:

- SLOC de las clases de implementación: 181
- SLOC de las clases que contienen los *tests*: 379

[1] https://site.mockito.org/

Destacar que el número de SLOC indicado anteriormente, pertenece únicamente a las clases que implementan los repositorios y servicios del sistema (que son sobre los que se ejecutan estas pruebas). El número total de SLOC del proyecto en su conjunto es 2629.

```
1   @Mock
2   AutomatismJpaRepository automatismJpaRepository;
3
4
5   private void mockInit() {
6
    when(this.automatismJpaRepository.findByRequestCode("Request
    Code")).thenReturn(this.getAutomatismEntity());
7       when(this.automatismJpaRepository.findByRequestCode("More
    Than One")).thenReturn(this.getMoreThanOneAutomatismEntity());
8
    when(this.automatismJpaRepository.findByRequestCode("Request
    Code Not Existent")).thenReturn(null);
9       when(this.automatismJpaRepository.save(any(
    AutomatismEntity.class))).thenReturn(
    this.getAutomatismEntity().get(0));
10  }
11
12  @Before
13  public void init() {
14      this.mockInit();
15      this.automatismRepository = new
    DefaultAutomatismRepository(this.automatismJpaRepository);
16  }
```

Fragmento de código 7.1: Ejemplo de uso de Mockito para las pruebas unitarias

7.2 Pruebas de integración

Las pruebas de integración son esenciales para analizar el correcto funcionamiento del sistema de forma conjunta. En ellas ya no se prueban fragmentos de código concretos, sino que abarcan todo un flujo completo.

Para su elaboración, se ha optado por realizar llamadas a las APIs disponibles mediante Postman[2] y comprobar que las respuestas obtenidas son las esperadas. En la Figura 7.1 se puede ver un ejemplo de estas pruebas.

[2] https://www.postman.com/

7.3 Pruebas no funcionales

Estas pruebas, a diferencia de las anteriores, no se centran en funcionalidades específicas del sistema, sino que analizan el rendimiento del mismo en diferentes entornos. Permiten conocer qué riesgos corre y proporcionan información sobre su comportamiento.

Se han realizado pruebas de rendimiento sobre los distintos sistemas resolutores. Dichas pruebas consisten en recoger (empleando la herramienta de *ticketing* del proyecto) todos los *tickets* que han sido automatizados por cada uno de ellos y, a partir de su fecha de creación y fecha de cierre, obtener la mediana del tiempo de procesamiento de todos ellos. Finalmente, se compara dicho valor con el correspondiente a los *tickets* que no han sido automatizados. Los resultados de estas pruebas se pueden observar en la Tabla 8.1, y su análisis en la sección 8.1.

Mencionar también que no se han realizado pruebas de carga o estrés específicas para el sistema, ya que la empresa confía en la alta disponibilidad que asegura el servidor de colas utilizado. Éste, como se ha explicado anteriormente en la sección 4.3, es capaz de migrar las conexiones y tareas hacia otros servidores en caso de caída de uno de estos, permitiendo así un funcionamiento continuo del sistema.

Finalmente, se ha configurado el despliegue de forma que se realice simultáneamente en dos nodos distintos con un balanceador de carga, que irá repartiendo el trabajo entre ambos nodos. De esta forma, se evita una sobrecarga innecesaria en cada uno de ellos y, además, cada uno podría actuar como "respaldo" del otro, cubriendo su trabajo en caso de caída.

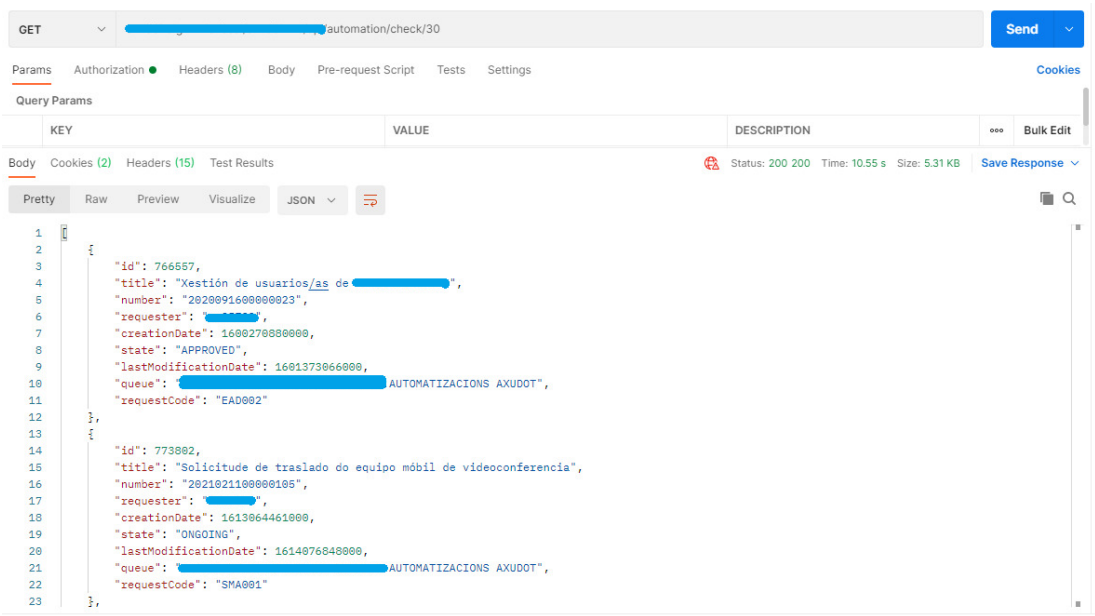


Figura 7.1: Ejemplo de llamada al servicio de monitorización desde Postman

Capítulo 8

Conclusiones

EN este capítulo se analizan los resultados obtenidos, valorando la medida en las que estos cumplen los requisitos del proyecto. Además, se expone la relación del mismo con los conocimientos adquiridos en el grado y su aplicación. Finalmente, se presentan las probabilidades de trabajo futuro sobre el sistema.

8.1 Resultados

Tras varios meses de desarrollo, se considera que los objetivos establecidos al comienzo del proyecto se han cumplido. Se ha conseguido diseñar e implementar un sistema plenamente funcional y adaptado a las necesidades de la empresa y a los requisitos de los clientes. Además, para comprobar la eficacia del mismo, se realizan análisis mensuales sobre los procesamientos de los distintos sistemas resolutores.

Dichos análisis consisten en la recogida de todos los *tickets* destinados a cada uno de los resolutores, agrupándolos de forma que se puedan calcular las medianas del tiempo dedicado (en horas) del número total de los creados desde AXUDOT, de los automatizados por el sistema, y, finalmente, de los no automatizados. En la Tabla 8.1 se puede observar el desglose de estos valores, correspondientes a los datos del mes de abril de 2021 para los sistemas resolutores 1 y 2.

Como se puede observar, la ventaja que se obtiene con los procesamientos automáticos es muy notable, pudiendo alcanzar diferencias tan grandes en algunos momentos como en el sistema resolutor 1; aunque lo habitual son diferencias más similares a las vistas en el resolutor 2. Esto confirma que el objetivo inicial se ha alcanzado, y se puede concluir que el trabajo ha sido un éxito.

8.2 Aplicación de las competencias del grado

La realización de este proyecto le ha permitido al alumno aplicar una importante cantidad de conocimientos adquiridos a lo largo del grado. A continuación, se exponen los principales ejemplos de ello.

Primeramente, destacar que los conocimientos y habilidades desarrollados en la asignatura *Metodologías de Desarrollo* han sido fundamentales para una correcta y rápida integración en la metodología adoptada en el proyecto (Scrum) y para una correcta ejecución de la misma.

Así mismo, los conocimientos conseguidos gracias a la asignatura *Herramientas de Desarrollo* también han contribuido a la rápida adaptación al proyecto, ya que, gracias a estos, el alumno no ha necesitado formación extra en la mayoría de las herramientas empleadas, y ha podido hacer un uso eficiente de ellas.

Por otro lado, las asignaturas de *Diseño Software*, *Análisis de Requisitos* y *Arquitectura del Software* han sido la base sobre la que se han tomado las decisiones de diseño y arquitectura del sistema.

En lo referido al modelo de datos, hay que destacar la importancia que han tenido las asignaturas de *Bases de Datos* y *Bases de Datos Avanzadas*, gracias a las cuales se ha podido diseñar un modelo de datos adecuado para el proyecto, junto con unas consultas eficientes.

A la hora de planificar el trabajo a abordar y realizar el seguimiento del mismo, se han puesto en práctica las habilidades aprendidas en la materia *Gestión de Proyectos*.

Finalmente, la experiencia de diseñar e implementar servicios REST y de trabajar con sistemas multicapa que proporciona la materia de *Internet y Sistemas Distribuidos*, junto con las habilidades en *frameworks* modernos como Spring adquiridas en *Programación Avanzada*, han sido fundamentales a la hora de afrontar y desarrollar este proyecto.

8.3 Trabajo futuro

Con la finalización del proyecto, se plantean distintas opciones para mejorar y ampliar las funcionalidades del sistema:

- Integración de nuevos sistemas resolutores con el fin de automatizar un mayor número de solicitudes.
- Implementar aprobaciones automáticas de solicitudes.
- Diseñar un servicio de trazabilidad completa de las solicitudes, que registre todos los pasos que siguen y pueda consultarse en tiempo real.
- Estudiar la inclusión de nuevas reglas de aprobación que optimicen la automatización y se acepten un mayor número de tickets.

Además, cabe destacar que, debido a la satisfacción del cliente con el resultado final, este ha propuesto un plan de trabajo futuro de varios años en el que se hará uso de los mecanismos implementados en este proyecto.

Sistema resolutor 1	Volumen	Tiempo mediano de resolución (horas)
Creados en AXUDOT	134	2,86
Automatizados por el sistema	128	2,59
No automatizados	7	192

Sistema resolutor 2	Volumen	Tiempo mediano de resolución (horas)
Creados en AXUDOT	127	11,3
Automatizados por el sistema	54	3,96
No automatizados	73	20,33

Tabla 8.1: Análisis de tiempos de resolución

Lista de acrónimos

API Application Programming Interface. 19, 34, 38, 40–42, 58

CAU Centro de Atención al Usuario. 26, 51

DTO Data Transfer Object. 34, 36, 37, 40, 50, 57

JMS Java Message System. 18, 19, 34

JPA Java Persistence API. 5, 40

MoM Message-oriented Middleware. 17, 18

POM Project Object Model. 6

REST REpresentational State Transfer. 38, 40–42, 62

SLOC Source Lines Of Code. 57, 58

Bibliografía

[1] "Java," consultado el 5 de mayo de 2021. [En línea]. Disponible en: https://www.java.com/es/

[2] C. Bauer, G. King, and G. Gregory, *Sistemas de bases de datos : un enfoque práctico para diseño, implementación y gestión*, 2nd ed. Manning Publications, 2016.

[3] C. Walls, *Spring in action*, 3rd ed. Manning Publications, 2011.

[4] "Apache maven project," consultado el 7 de abril de 2021. [En línea]. Disponible en: https://maven.apache.org/index.html

[5] "Apache activemq," consultado el 13 de abril de 2021. [En línea]. Disponible en: https://activemq.apache.org/components/artemis/documentation/latest/book.pdf

[6] "Intellij idea," consultado el 13 de abril de 2021. [En línea]. Disponible en: https://www.jetbrains.com/es-es/idea/

[7] "Sql developer," consultado el 13 de abril de 2021. [En línea]. Disponible en: https://www.oracle.com/es/database/technologies/appdev/sqldeveloper-landing.html

[8] "Apache subversion," consultado el 13 de abril de 2021. [En línea]. Disponible en: https://subversion.apache.org/

[9] "Docker," consultado el 13 de abril de 2021. [En línea]. Disponible en: https://docs.docker.com/get-started/overview/

[10] "Docker compose," consultado el 13 de abril de 2021. [En línea]. Disponible en: https://docs.docker.com/compose/

[11] "Jenkins," consultado el 14 de abril de 2021. [En línea]. Disponible en: https://www.jenkins.io/doc/

[12] "Redmine," consultado el 14 de abril de 2021. [En línea]. Disponible en: https://www.redmine.org/

[13] K. Schwaber and M. Beedle, *Agile Software Development with Scrum*, 1st ed. Prentice Hall, 2002.

[14] "Scrum guide spanish," consultado el 22 de abril de 2021. [En línea]. Disponible en: https://scrumguides.org/docs/scrumguide/v2016/2016-Scrum-Guide-Spanish.pdf#zoom=100

[15] "Xvi convenio colectivo estatal de empresas de consultoría y estudios de mercado y de la opinión pública." [En línea]. Disponible en: https://www.boe.es/diario_boe/txt.php?id=BOE-A-2009-5688

[16] L. Richardson and S. Ruby, *RESTful Web Services*, 1st ed. O'Reilly, 2007.